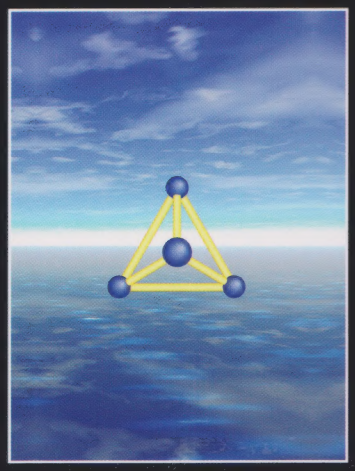
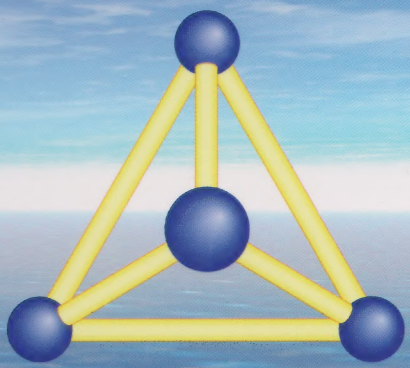
The Open University

MT365 Graphs, networks and design



Graphs 4

Graphs and computing



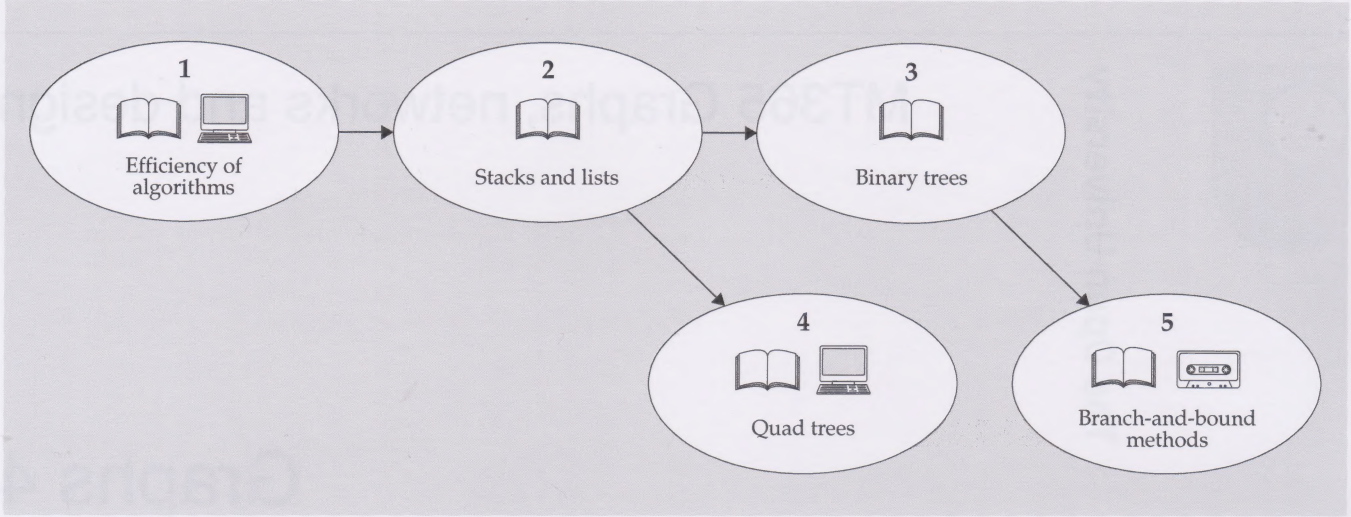
The Open University

MT365 Graphs, networks and design

Graphs 4

Graphs and computing

Study guide



The material in Sections 1 and 2 is introductory to the rest of the unit, and you should not spend long on the details of these sections. The most important sections of this unit are Sections 3, 4 and 5, and these sections will probably take up most of your time.

There are computer activities associated with Sections 1 and 4.

Section 5 includes two audio-tape sessions.

The Open University, Walton Hall, Milton Keynes, MK7 6AA.

First published 1995. Second edition 2009.

Printed and bound by Page Bros, Norwich.

ISBN 978 0 7492 5425 4

2.1

Contents

Introduction

The disciplines of graph theory and computer science have much in common. They both deal with *finite discrete* sets, and seek to solve problems concerning these by the use of algorithms. In graph theory the sets are those of the vertices and edges or arcs of a graph, digraph or network, together with any weights on the edges or arcs. In computer science the sets are the data stored in the computer's memory; these sets are made up of numbers, symbols or strings of symbols. The algorithms form the basis of the programs that manipulate the data in various ways.

The exchange of ideas and notation between the two disciplines takes place in both directions.

Computing in graph theory

Graph theory is one of a number of mathematical subjects that uses the idea of an algorithm. When we work through graph algorithms with pencil and paper, we are necessarily limited to small structures. But the growth of computer power has meant that we can also apply the ideas to large structures. Graph theory is rich in problems that can be adapted for the computer. In the computing activities in this course, we have seen how a graph or network can be drawn on the screen and manipulated, and how we can find properties of various graphs or networks by running appropriate algorithms.

Graphs in computer science

Graph theory supplies some of the more elegant structures used in computer science — in particular, that of a *tree*. Also, although computer algorithms may look different from the graph algorithms we have described in the course, the principle behind them is the same. Furthermore, simple algorithms for graph structures often lead to an understanding of the design of computer algorithms.

Complexity theory

Another link between graph theory and computer science is provided by *complexity theory*. This is the study of how much time algorithms take and how large their inputs can be while still enabling an answer to be obtained in a reasonable amount of time. In graph theory, where many algorithms are relatively slow, we are mainly interested in the size of a graph that an algorithm can handle in a reasonable time. In computer science, where the majority of algorithms are designed to search and sort large sets of data, we are mainly concerned with improving the speed of such searches and sorts.

In Section 1, *Efficiency of algorithms*, we investigate the speed of algorithms, and explain what is meant by saying that an algorithm has a *time complexity function* of a certain *order*.

In Section 2, *Stacks and lists*, we introduce two important ways of storing data on a computer. We then present some important computer algorithms and calculate the orders of their time complexity functions.

Section 3, *Binary trees*, is concerned with various algorithms for searching trees; in particular, we discuss the important ideas of *depth-first search* and *breadth-first search*, and the orders of their time complexity functions.

In Section 4, *Quad trees*, we discuss some connections between images on computer screens and certain rooted trees. We also summarize the discussion of the orders of the time complexity functions of the algorithms discussed in the unit.

In Section 5, *Branch-and-bound methods*, we show how tree-searching algorithms can be used to solve various types of problem. These include the *knapsack problem* — that of trying to pack as many items into a knapsack so as to maximize their total value without exceeding an overall weight limit — and the *travelling salesman problem*.

1 Efficiency of algorithms

The speed at which a computer program is executed (independent of the computer used) depends on the efficiency of the algorithms it uses.

Recall the definition of an algorithm.

Recall, from the *Introduction* unit, that the *efficiency* of an algorithm is a measure of the time it takes to solve a problem.

Definition

An **algorithm** is a systematic step-by-step procedure consisting of:

- a description of appropriate input data;
- a finite, ordered list of instructions, to be carried out one at a time;
- a STOP instruction, to indicate when the procedure is complete;
- a description of appropriate output data.

This definition appeared in Section 4 of the *Introduction* unit.

The STOP instruction must be reached after a finite number steps.

To calculate the efficiency (or speed) of an algorithm, we must decide on a measure of the time taken by each instruction. Some instructions take longer than others. To make a start in our calculations, we assume that the most time-consuming instruction in any algorithm is one that compares two items of data to see whether they are the same. We say that *one comparison instruction* in an algorithm takes up *one time unit*. We ignore the time taken by other instructions, on the assumption that they do not contribute a significant amount to the overall time taken to complete the algorithm. The problem of calculating the efficiency of an algorithm thus becomes one of finding a function that determines the number of comparisons made by the algorithm.

The assumptions here are based on empirical evidence.

1.1 Time complexity functions

To determine an appropriate function for a given algorithm, we first need to specify the size of the input to the algorithm. We take this to be the non-negative integer n that gives the number of items in the set of input data (such as the number of vertices of a graph, or the number of items in a data store). We also need to specify the steps of the algorithm as precisely as possible, so that we can work out precisely when comparisons are made. We can then use this information to calculate a formula for the non-negative real number $T(n)$ that gives the total number of time units taken for an input of size n.

We assume that $T(n)$ is non-negative for all values of n, since it makes no sense to consider a negative number of comparisons.

Sometimes we can specify exactly how many time units are used for an input of size n, and sometimes we can find an estimate of the time taken. Usually, however, we have to settle for a formula $T(n)$ that gives the *maximum* amount of time taken for an input of size n.

Definition

For an algorithm, the maximum time taken to process any input of size n, at 1 time unit per comparison made, is called the **time complexity function** for the algorithm and is denoted by $T(n)$.

There is a similar function, the **space complexity function** $S(n)$, that gives the maximum *space* required in a computer's memory for storing the data used by an algorithm. Some algorithms are very 'hungry' when they are executed, in that they use a lot of space. Also, several algorithms rely on the data being stored in a certain way so that it can be accessed efficiently, and some ways of storing data use more space than others. The space complexity function is important in many situations, but we shall not consider it in any depth. However, we shall consider a number of ways to store data in the memory of a computer, and these can have a direct bearing on the speed at which certain algorithms perform.

Example 1.1

Suppose that we store a list of positive integers by writing them on a tape divided into numbered cells and we want to devise an algorithm to determine whether a given number is in the list. For example, we could store the list 1, 7, 3, 5, 8, 11 in the two ways shown in the margin. Suppose that we want to search the list to see whether it contains the number 9.

11	
10	
9	
8	
7	
6	11
5	8
4	5
3	3
2	7
1	1

(a)

On tape (a), we write the six numbers in cells 1 to 6. An algorithm that searches for 9 when the list is stored as in (a) starts at cell 1, checks whether 9 is stored in it, then repeats this checking process for each cell in numerical order until cell 6 is checked. In this case, the number 9 is not found. This worst-case instance takes $n = 6$ comparisons for our list of $n = 6$ numbers. Hence, for the general case of a list of n numbers, the algorithm has a time complexity function $T(n) = n$.

11	11
10	–
9	–
8	8
7	7
6	–
5	5
4	–
3	3
2	–
1	1

(b)

On tape (b), we set aside the cells numbered 1 to 11 and then write each number in the list in the cell with the same number, 1 in cell 1, 3 in cell 3, and so on, putting dashes in the unused cells. An algorithm that searches for 9 when the list is stored as in (b) just checks whether 9 is stored in cell 9. Thus only one comparison is needed, and this is also true for the general case of a list of n numbers. Hence the time complexity function for the algorithm is $T(n) = 1$.

The second algorithm, used on a store of the type used on tape (b), is much more efficient than the first algorithm, used on a store of the type used on tape (a). However, the first type of store makes more efficient use of the cells on the tape than the second type of store. In general, a store of type (a) has space complexity function $S(n) = n$, where n is number of items stored, whereas a store of type (b) has space complexity function $S(n) = m$, where m is the largest number in the list and where, for a list of n (different) positive integers, we must have $m \geq n$. For example, for a list of 6 positive integers, the largest of which is 1023, a store of type (a) uses just 6 cells whereas one of type (b) uses 1023. ■

We know the time complexity functions for many of the algorithms in computer science and graph theory — some exactly, and most of the rest approximately. The time complexity function of an algorithm tells us whether we can expect an algorithm to give a result in a reasonable time. In particular, if the function gives a 'large' $T(n)$ value for a 'small' value of n, then we know we should use the algorithm on 'small' inputs or not at all. It also enables us to decide whether we should look for a more efficient algorithm for the problem.

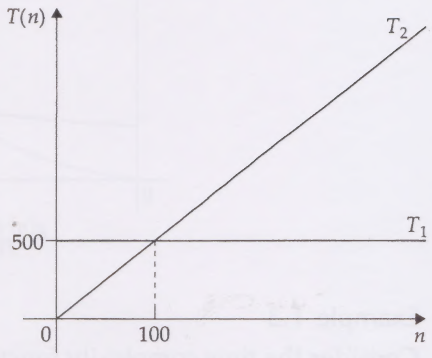
To compare the efficiency of two algorithms, we compare their time complexity functions. This is not as simple as it sounds, since — as we saw in the *Introduction* unit — the comparative values of such functions may change as the size of the input changes.

Example 1.2

Suppose that an algorithm has time complexity function $T_1(n) = 500$ time units. This algorithm produces a result after 500 time units, no matter what the size of the input. Now suppose that another algorithm for the

same problem has time complexity function $T_2(n) = 5n$. Here the time taken is directly proportional to the size of the input. For a small input, of size $n = 10$ say, the first algorithm churns away for 500 time units while the second delivers after 50 time units, and is clearly faster for an input of this size. However, the superiority of the first algorithm becomes apparent when the input is 'large'. For $n = 1000$, the first algorithm still takes 500 time units, while the second algorithm takes 5000 time units, and is much slower for inputs of this size or larger.

In fact, we can see from the diagram in the margin that the first algorithm is faster than the second — the graph of T_2 lies above the graph of T_1 — whenever n is greater than 100. Thus, if we deal with inputs whose size is always less than 100, then we should use the second algorithm; however, if n is likely to be larger than 100, then we should use the first algorithm. We summarize the algorithm times for different values of n in the table below.



n	< 100	100	> 100
$T_1(n) = 500$	500	500	500
$T_2(n) = 5n$	< 500	500	> 500

■

Problem 1.1

Draw up a table comparing the values of the time complexity functions $T_1(n) = 100$, $T_2(n) = 10n$ and $T_3(n) = n^2$, for

(a) $n = 5$; (b) $n = 10$; (c) $n = 20$.

Comment on your results.

Generally, when comparing the efficiency of algorithms for use with a computer, we want to make the comparison only for 'large' input sizes n, where the meaning of 'large' depends on the context of the problem. For example, for the algorithms above, we could sensibly take 'large' to mean 'greater than the number $N = 100$'. We say that we compare the *asymptotic behaviour* of the functions.

1.2 Order of a time complexity function

To compare the time complexity functions of algorithms in general, we adopt a notation that is widely used in both pure and applied mathematics, called the big-oh notation.

Definitions

Let $T(n)$ be a time complexity function, and let $g(n)$ be a function for which there exists both a positive constant c and a (large enough) number N such that

$$T(n) \leq c.g(n), \quad \text{for all } n \geq N. \tag{1.1}$$

Then $g(n)$ is said to **(asymptotically) dominate** $T(n)$, and $T(n)$ is said to be **(asymptotically) dominated by** $g(n)$.

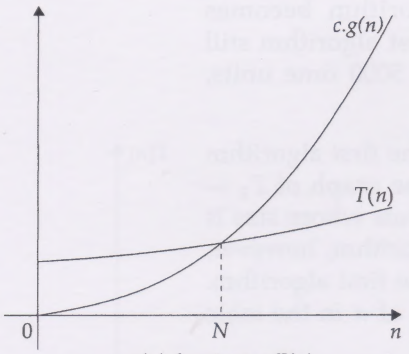
The set of all functions that $g(n)$ dominates is denoted by $\boldsymbol{O(g(n))}$.

If $g(n)$ dominates $T(n)$, we say that $T(n)$ is $O(g(n))$, meaning $T(n)$ is in the set $O(g(n))$.

This definition applies to any function $T(n)$ and not just to time complexity functions.

The set $O(g(n))$ — pronounced big-oh of $g(n)$ — was proposed in 1976 by Donald Knuth. He attributes the big-oh notation to P. Bachmann, who devised it to deal with approximations.

Inequality 1.1 is easier to understand in terms of the graphs. It means that we are able to adjust the graph of $g(n)$, by multiplying by some positive constant c, such that eventually (for all n greater than some number N) the graph of $c.g(n)$ lies above that of $T(n)$, and remains above it.



$g(n)$ dominates $T(n)$
$T(n)$ is $O(g(n))$

Example 1.3

Consider the time complexity functions $T_1(n) = 500$ and $T_2(n) = 5n$.

The function $T_1(n)$ is $O(1)$, since inequality 1.1 holds for $T_1(n)$ with $g(n) = 1$, $c = 500$ and $N = 0$:

$$500 = T_1(n) \leq c.g(n) = 500.1 \quad \text{for all } n \geq 0.$$

The function $T_1(n)$ is also $O(n)$, since inequality 1.1 also holds for $T_1(n)$ with $g(n) = n$, $c = 500$ and $N = 1$, say. Similarly $T_1(n)$ is $O(n^k)$ for any integer $k \geq 2$.

The function $T_2(n)$ is $O(n)$, since inequality 1.1 holds for $T_2(n)$ with $g(n) = n$, $c = 5$ and $N = 0$:

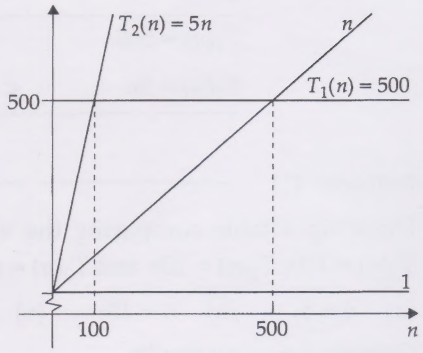
$$5n = T_2(n) \leq c.g(n) = 5.n \quad \text{for all } n \geq 0.$$

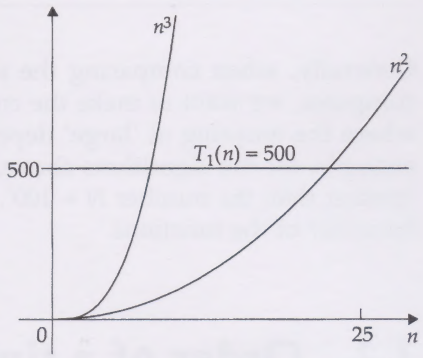
The function $T_2(n)$ is also $O(n^2)$, since inequality 1.1 also holds for $T_2(n)$ with $g(n) = n^2$, $c = 5$ and $N = 1$, say. Similarly $T_2(n)$ is $O(n^k)$ for any integer $k \geq 3$.

However, $T_2(n)$ is not $O(1)$, since

$$5n = T_2(n) \leq c.g(n) = c.1$$

holds only when $5n \leq c$, that is when $n \leq c/5$. So, however large we make c, inequality 1.1 will *not* hold when $n > c/5$. We *cannot* find an N for which inequality 1.1 holds for all $n \geq N$. ■



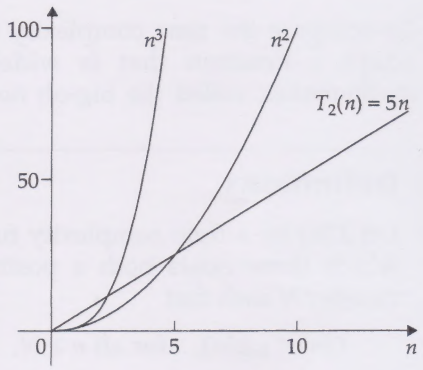


Problem 1.2

Consider the quadratic time complexity function $T(n) = 2n^2 + 4n + 3$.

(a) Show that $T(n)$ is not dominated by 1 and n (i.e. is not $O(1)$ or $O(n)$) but is dominated by n^2 and n^3 (i.e. is $O(n^2)$ and $O(n^3)$).

(b) Show that $T(n)$ dominates 1, n and n^2 but does not dominate n^3.



We can generalize the results of Problem 1.2 to show that for any polynomial time complexity function $T(n) = pn^m + qn^{m-1} + \ldots + r$ $(p > 0)$:

(a) n^k $(k < m)$ does not dominate $T(n)$;

(b) n^k $(k \geq m)$ dominates $T(n)$;

(c) $T(n)$ dominates n^k $(k \leq m)$;

(d) $T(n)$ does not dominate n^k $(k > m)$.

Hence n^m is the *only* function of the form n^k, for k a non-negative integer, that both dominates and is dominated by $T(n) = pn^m + qn^{m-1} + \ldots + r$ $(p > 0)$. This prompts the following definition.

Definition

If a function $g(n)$ dominates a time complexity function $T(n)$ and $T(n)$ also dominates $g(n)$ then $T(n)$ is said to have **order $O(g(n))$**, and $T(n)$ and $g(n)$ are said to have the **same order of magnitude**.

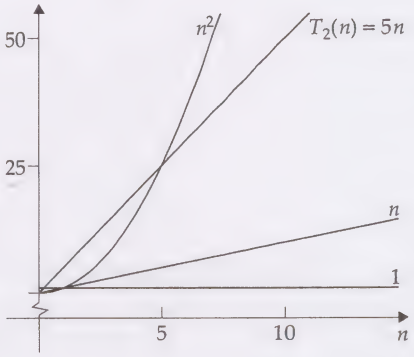
Again, this definition applies to any function $T(n)$ and not just to time complexity functions.

Therefore $T(n) = pn^m + qn^{m-1} + ... + r$ $(p > 0)$ has order $O(n^m)$. Roughly speaking, if two functions have the same order of magnitude, their graphs behave in roughly the same way.

Example 1.4

The time complexity function $T_2(n) = 5n$ has order $O(n)$. The graphs of $5n$ and n behave roughly the same way, in that they are straight lines which slope upwards.

However, since $T_2(n)$ does not have order $O(1)$ or order $O(n^2)$, the graph of $5n$ behaves very differently from those of 1 and n^2. The graph of 1, although a straight line, has zero slope. The graph of n^2, although it slopes upwards, is not a straight line. ■



We generally consider two algorithms to have roughly the same efficiency if their time complexity functions are both of order $O(g(n))$ for some function $g(n)$.

1.3 A hierarchy of orders

Consider the inequality $1 \leq n$, which holds for all $n \geq 1$. If we multiply both sides of this by n (≥ 1), we obtain $n \leq n^2$. Multiplying both sides by n again, gives $n^2 \leq n^3$. And so on. Hence we have

$$1 \leq n \leq n^2 \leq n^3 \leq n^4 \leq ... \leq n^k \leq ... \quad \text{for all } n \geq 1.$$

From this we can deduce that any function $T(n)$ dominated by 1 is dominated by n, since

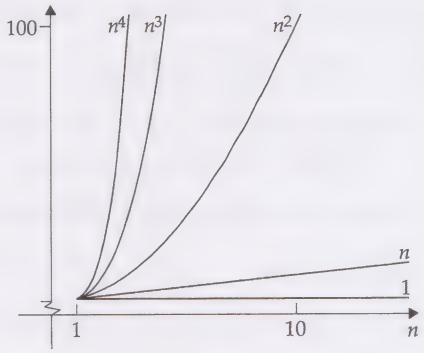
$$\text{if } T(n) \leq c.1 \text{ then } T(n) \leq c.n \quad \text{for all } n \geq 1.$$

Similarly any function $T(n)$ dominated by n is dominated by n^2, since

$$\text{if } T(n) \leq c.n \text{ then } T(n) \leq c.n^2 \quad \text{for all } n \geq 1.$$

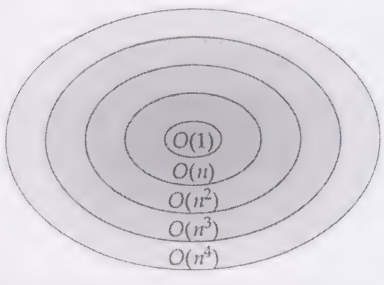
And so on. We therefore obtain the following set inclusions, known as a **hierarchy of orders**:

$$O(1) \subset O(n) \subset O(n^2) \subset O(n^3) \subset O(n^4) \subset ... \subset O(n^k) \subset ...$$



$A \subset B$ means that the set A is contained in the set B.

The set inclusions are *proper*, in that equality of sets does not occur. For example, we have seen that the function $T_2(n) = 5n$ is in $O(n)$ but not in $O(1)$.



Every polynomial time complexity function $T(n)$ can be **placed** in just *one* of the sets in the hierarchy, namely the set whose *defining function* — $1, n, n^2, n^3, n^4, ..., n^k, ...$ — has the same order of magnitude as $T(n)$. For example, we have seen that $T(n) = pn^m + qn^{m-1} + ... + r$ $(p > 0)$ is of order $O(n^m)$ and so we place it in the set $O(n^m)$.

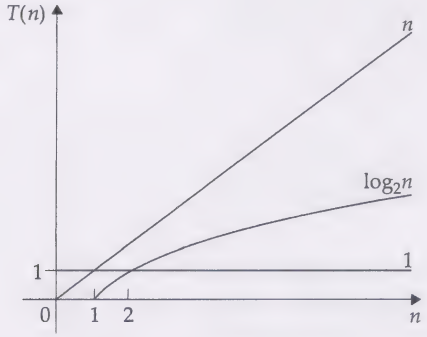
From the way we constructed the hierarchy, any time complexity function placed in the set $O(n^i)$ is dominated by any time complexity function placed in the set $O(n^j)$, where $j > i$. This means that any algorithm with time complexity function of order $O(n^i)$ is faster (more efficient), *for large enough* n, than any algorithm with time complexity function $O(n^j)$, where $j > i$. Thus, any algorithm with a constant time complexity function is faster than one with a linear time complexity function, any algorithm with a linear time complexity function is faster than one with a quadratic time complexity function, and so on, with the proviso 'for large enough n' in each case. In other words, the *further to the left* in the hierarchy we can place a time complexity function, the *faster* is the corresponding algorithm — for large enough n.

The *constant functions* are those of the form $T(n) = r$, for some number r; the *linear functions* are those of the form $T(n) = qn + r$, for some numbers $q \neq 0$ and r; the *quadratic functions* are those of the form $T(n) = pn^2 + qn + r$, for some numbers $p \neq 0, q$ and r.

We can add other sets to the above hierarchy. For instance, there are computer algorithms with time complexity functions of the same order of magnitude as the simple function $\log_2 n$. Comparing the graph of $\log_2 n$ with those of 1 and n, we see that, for $n > 2$, $1 < \log_2 n < n$. Therefore we can deduce that the set $O(\log_2 n)$ lies between $O(1)$ and $O(n)$ in the hierarchy.

If $n = 2^k$, then $k = \log_2 n$; for example, $\log_2 1 = 0$, $\log_2 2 = 1$, $\log_2 4 = 2$, ... , $\log_2 32 = 5$, and so on.

$$O(1) \subset O(\log_2 n) \subset O(n) \subset \ldots$$



The set $O(\log_2 n)$ is a particularly useful one to add to the hierarchy since all logarithm functions — no matter what their base — have order $O(\log_2 n)$. This can be shown using the following general result about logarithm functions:

$$\log_b n = \frac{\log_a n}{\log_a b} \qquad (1.2)$$

In general, if $n = a^k$ then $\log_a n = k$.

Example 1.5

Consider the function $\log_3 n$. Using equation 1.2, we have

$$\log_3 n = \frac{\log_2 n}{\log_2 3} \leq 1.\log_2 n \qquad \text{(since } 1/\log_2 3 \approx 0.6 < 1\text{)}.$$

So $\log_2 n$ dominates $\log_3 n$. Also, again using equation 1.2, we have

$$\log_2 n = \log_2 3 \times \log_3 n \leq 2.\log_3 n \qquad \text{(since } \log_2 3 \approx 1.6 < 2\text{)}.$$

So $\log_3 n$ dominates $\log_2 n$. Therefore $\log_3 n$ is of order $O(\log_2 n)$. ■

Problem 1.3

Locate the set $O(n\log_2 n)$ in the above hierarchy.

We can keep adding sets to the hierarchy in the above way to produce the following hierarchy of orders showing how the more common big-oh sets fit in.

Definition

The **order hierarchy** of common big-oh sets is

$$O(1) \subset O(\log_2 n) \subset O(n) \subset O(n\log_2 n) \subset O(n^2) \subset \ldots \subset O(n^k) \subset \ldots$$
fast

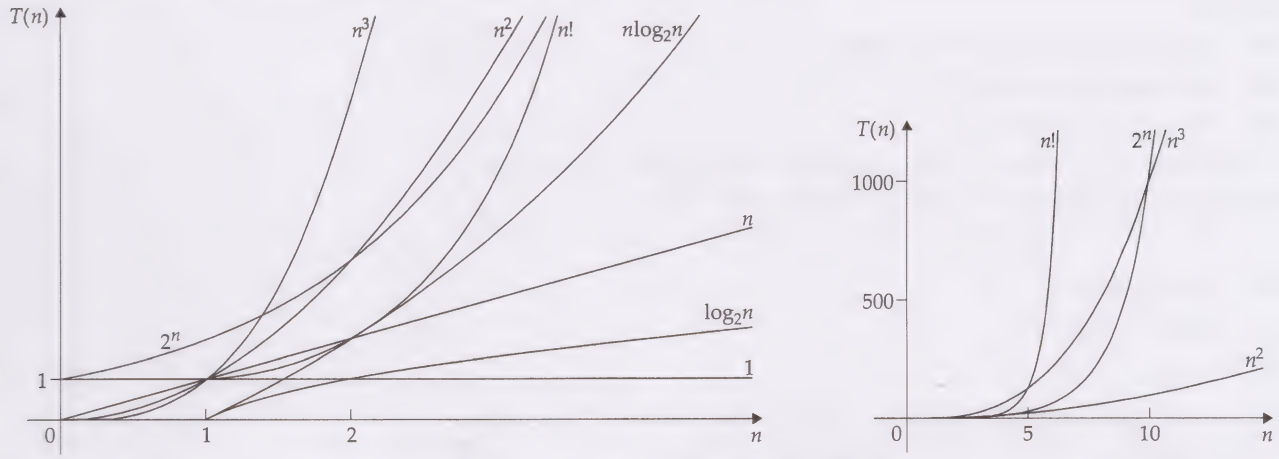
$$\ldots \subset O(2^n) \subset O(n!).$$
slow

The time complexity function $T(n)$ of an algorithm is **placed** in the set $O(g(n))$ in the hierarchy if $T(n)$ has order $O(g(n))$. The *further to the left* in the hierarchy $T(n)$ is placed, the faster is the algorithm (for large enough n).

The 'faster' algorithms are those whose time complexity functions are placed in the sets with polynomial or logarithmic defining functions — these are the *polynomial-time algorithms*. The 'slower' algorithms are those whose complexity functions are placed in the sets $O(2^n)$ or $O(n!)$ at the right of the hierarchy — these are the *exponential-time algorithms*. The graphs of the defining functions for the sets in the hierarchy are

The terms *polynomial-time* and *exponential-time* were defined in the *Introduction* unit.

shown below. These give a good indication of the relative speeds of the corresponding algorithms. Notice, in particular, how the functions 2^n and $n!$ increase very rapidly for small increases in n.



All the time complexity functions of the algorithms in this course can be placed in one of the sets in the order hierarchy. In particular, we can place in the hierarchy a time complexity function that is the sum of constant multiples of the defining functions. The following example illustrates how this can be done.

Example 1.6

Consider an algorithm with time complexity function

$$T(n) = 3n\log_2 n + 5n.$$

We want to place $T(n)$ in the order hierarchy.

The terms $3n\log_2 n$ and $5n$ are easily placed in the appropriate sets in the hierarchy: we already know that $5n$ is of order $O(n)$ and it is easy to see that $3n\log_2 n$ is of order $O(n\log_2 n)$. Hence n dominates $5n$ and $n\log_2 n$ dominates $3n\log_2 n$ — that is, there exist positive constraints c and d and numbers N_1 and N_2 so that:

$$5n \le c.n \quad \text{for all } n \ge N_1;$$

$$3n\log_2 n \le d.n\log_2 n \quad \text{for all } n \ge N_2.$$

We also know that $n \le n\log_2 n$. Therefore, if N is the larger of N_1 and N_2, then for all $n \ge N$

$$\begin{aligned} 3n\log_2 n + 5n &\le d.n\log_2 n + c.n \\ &\le d.n\log_2 n + c.n\log_2 n \\ &= (d+c).n\log_2 n. \end{aligned}$$

In other words $n\log_2 n$ dominates $3n\log_2 n + 5n$.

Also,

$$n\log_2 n \le 3n\log_2 n \le 3n\log_2 n + 5n \quad \text{for all } n \ge 1,$$

and so $3n\log_2 n + 5n$ dominates $n\log_2 n$.

Hence $T(n) = 3n\log_2 n + 5n$ is of order $O(n\log_2 n)$. ■

The above type of analysis can be performed for any time complexity function consisting of a sum of constant multiples of defining functions from the order hierarchy. Its success rests on two key factors:

(a) a constant multiple of a defining function has the same order as the defining function;

(b) a sum of constant multiples of defining functions is dominated by the defining function whose set lies furthest to the right in the order hierarchy.

These facts form the basis of the following simple procedure for finding the order of a time complexity function.

Placing a time complexity function in the order hierarchy

Given a time complexity function $T(n)$ consisting of the sum of one or more terms:

- take the base of each logarithm to be 2;
- take each coefficient to be 1;
- take each constant term to be 1.

Then the term $g(n)$ whose set lies furthest to the right in the order hierarchy is the dominant term, and $T(n)$ has order $O(g(n))$.

Example 1.6 continued

Using the above procedure on

$$T(n) = 3n\log_2 n + 5n,$$

we first reduce each coefficient to 1 to obtain $n\log_2 n + n$. Then, since $O(n) \subset O(n\log_2 n)$, we deduce that $T(n)$ is of order $O(n\log_2 n)$. ■

Problem 1.4

Two algorithms for a problem have the following time complexity functions:

$$T_1(n) = 9n^3 + 5n + 3\log_{10} n$$

$$T_2(n) = 1000\log_2 n + 2n^2 + 100$$

Determine the order of each, and hence deduce which algorithm is faster.

If we wish to, we can add further sets to the hierarchy. For example, it is easy to deduce that 2^n is dominated by 3^n, which is dominated by 4^n, ..., which is dominated by $n!$, giving the following additions to the right of the hierarchy:

$$O(2^n) \subset O(3^n) \subset O(4^n) \subset \ldots \subset O(k^n) \subset O(n!).$$

Problem 1.5

(a) Locate the set $O(n^2\log_2 n)$ in the order hierarchy.

(b) Hence determine which algorithm is faster, one with time complexity function $2n^2 + \log_3 n$ or one with time complexity function $4n^2\log_2 n$.

1.4 Computer activities

The computer activities for this section are described in the *Computer Activities Booklet*.

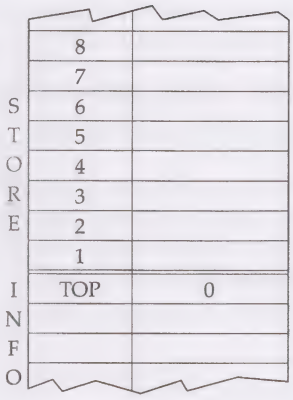
After studying this section, you should be able to:

- understand what is meant by the *time complexity function* for an algorithm;
- understand what is meant by (*asymptotic*) *domination*, the notation $O(g(n))$ and the *order of a time complexity function*;
- determine the order of a given time complexity function, and hence place the function in the *order hierarchy*;
- compare the efficiency (speed) of two or more algorithms, given their time complexity functions.

2 Stacks and lists

We now turn our attention to some basic algorithms of computer science and to determining their time complexity functions. Computer algorithms manipulate data stored in a computer. The simplest, and most frequently used, algorithms search the data for a given item or rearrange the data into some required order. The size of the input for such algorithms is just the number of items in the store.

A simple model for storing items is shown in the margin. It consists of a long tape divided into two distinct areas. The major area is the STORE, consisting of cells numbered 1, 2, 3, ..., as shown, giving each storage cell an *address*. We can choose a storage cell by specifying its address and then write an item of data into that cell, erase an item already there (leaving the cell blank), or overwrite an item — that is, first erase and then write. Examples of items that can be stored are numbers, letters of the alphabet, or strings of characters.



There are also INFORMATION cells, not part of the store, used to hold the address of a cell in the store or other useful information. These cells, shown below the double line on the tape, have labels that indicate the type of information that the cell holds.

The way we store the items on the tape has a bearing on what we can do with the data. We start by describing the simplest way of entering items onto the tape.

2.1 The stack data type

Suppose that we wish to enter into the store the following list of words, each word being considered as a separate item of data:

lion, tiger, aardvark, zebra, camel, dingo, lion, gorilla.

Note that the word *lion* occurs twice.

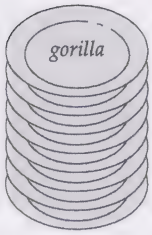
We write the items on the tape as we read them from the list. To begin with, the store is empty and there is one information cell, labelled TOP, with the number 0 in it. To enter the first item, we increase the number in TOP by 1, so that it now has 1 in it, and use this as the address of the store cell in which we write the item. In the cell with address 1 we write the item *lion*. We now repeat the procedure, increase the number in TOP by 1 (to 2), and write *tiger* in the cell with address 2. When we have entered all the items in this manner, the tape has the number 8 in the information cell TOP, and all the words written in the store cells as shown, with the last item *gorilla* entered in the cell with address 8.

8	
7	
6	
5	
4	
3	
2	
1	*lion*
TOP	1

8	*gorilla*
7	*lion*
6	*dingo*
5	*camel*
4	*zebra*
3	*aardvark*
2	*tiger*
1	*lion*
TOP	8

For the moment, we keep the method of storing items as simple as possible. We always start with the first item being stored at the cell with address 1, and the only information cell we have is TOP, which contains either the address at which the last item entered is stored, or 0 if no items are stored.

We call such a data store a **stack**. You can think of the items being stacked, one upon another, like a stack of dishes, each dish having a word written on it. In such a stack, the only word we can see is the one written on the top dish. This corresponds to the fact that, on the tape, the only address we have kept is the one contained in the information cell TOP.



Even with such a simple data store as a stack, there are several inherent *basic operations* that we can perform on the data.

We can determine the top (or last entered) item of the stack, by looking up the address held in TOP, going to the cell with this address and reading the item written in this cell. We write

TOP(s) = the top item of the stack s;

for example, for the stack s of words above, we have TOP(s) = *gorilla*.

We can add a further item to a stack s, thereby forming a new stack s'. We increase the address in the cell TOP by 1, go to the store cell with this new address and write the new item in it. The cell TOP now has the address of the top item of the new stack s'. We call this operation PUSH, since we 'push' a new item onto the old stack, thereby creating the new stack s'. We write

PUSH(item, s) = s',

and call s' the **push stack** of s.

We can also decrease a stack s by one element, by decreasing the address held in TOP by 1. This gives a new stack s', with one fewer item than s. We call this operation POP ('popping' an item off the stack). We write

POP(s) = s',

and call the stack s' the **pop stack** of s.

Problem 2.1

If s is the stack shown, draw the tape for each of the following:

(a) the stack PUSH(*iguana*, s);

(b) the stack POP(s);

(c) the stack POP(POP(s)).

4	
3	*aardvark*
2	*tiger*
1	*lion*
TOP	3

If a stack has only one element, then the address of its top element is 1 and after applying POP we end up with 0 in the cell TOP; as we have indicated, this means that there are no items in the store. In this case, we call the pop stack the **empty stack**; it is the unique stack with no items in it. We can think of the start of the process of entering items into a store as the empty stack waiting to have items pushed onto it. When we start to enter the words above, the first operation is PUSH(*lion*, empty stack).

Note that we cannot perform either of the operations POP or TOP on the empty stack, as each requires at least one item in the store. So, before we can apply them, we need to check that a stack is not empty — that is, that there is no 0 in the information cell TOP.

The above discussion leads to another basic operation on stacks — one that asks whether a given stack is empty. We call it ISEMPTYSTACK? and it looks at the number in the information cell TOP. If this number is 0, then it returns TRUE (the stack is empty); otherwise, it returns FALSE.

There is one further basic operation for stacks. Since we always start a stack from address 1, the number in the cell TOP is not only the address of the top item in the stack, but is also the number of items in the stack — we call this number the **depth** of the stack. We therefore have an operation DEPTH that returns the number in the cell TOP.

Problem 2.2

For the stack s of Problem 2.1:

(a) draw the tape as it would be after the operation

PUSH(*iguana*, POP(s));

(b) write down the item given by TOP(POP(s));

(c) write down the value of DEPTH(POP(s));

(d) describe how to apply the basic operations to s to determine the third item from the top of the stack.

A stack is the simplest way storing data on a tape; it has a surprising amount of structure to it. The store of data, together with the basic operations used to create it, change it and give information about it, is called a **data type**.

Definition

The **stack data type** consists of data stored in the above manner, together with the following basic operations:

TOP(s) = the top item in the stack s;

DEPTH(s) = the number of items in the stack s;

PUSH(item, s) = the stack created by pushing (adding) the item onto the top of the stack s;

POP(s) = the stack created by popping (removing) the top item from the stack s;

ISEMPTYSTACK? = TRUE if s is the empty stack, and FALSE otherwise.

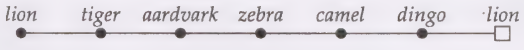
Thus a data type consists not only of the store of data, but also the basic operations that manipulate the store. The basic operations on the data are usually implemented on a computer. Any further operations that we wish to carry out must be described by an algorithm that makes use of the basic operations of the data type.

The stack data type is very simple, but this simplicity has some disadvantages. Because of the way that we have created this simple structure, all we can do is to keep track of the last item that we PUSH onto a stack, and this item is the only one that we can discard from the stack by using POP. For this reason, a stack is often referred to as a *LIFO stack* (Last In, First Out). You may think that we could delete the first item of a stack, since we know that its address is 1, but to do this we should have to move each other item down one address, since we insist that the resulting stack must start with its first item at address 1. To write an algorithm that does this, using just the basic operations, would be tedious. On the plus side, a stack makes efficient use of storage space. It stores n items in n cells on the tape, plus one cell for keeping information in.

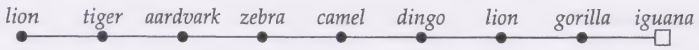
Since one unit of space corresponds to one cell on the tape, the space complexity function for storing n items in a stack is $S(n) = n + 1$, of order $O(n)$.

Instead of writing on a tape, we can represent a stack of n items as a rooted path graph P_n. The root vertex corresponds to the *top* item in the stack, and the other items correspond to the other vertices. For example, the path graph of our familiar stack of items is:

The operation POP corresponds to deleting the root vertex and its incident edge. The resulting **pop graph** represents the pop stack.



The operation PUSH corresponds to adding a new vertex and edge to the root of the old graph. The resulting **push graph** represents the push stack.



The operation ISEMPTYSTACK? corresponds to asking whether the graph has any vertices, and the operation DEPTH returns the number of vertices in the graph.

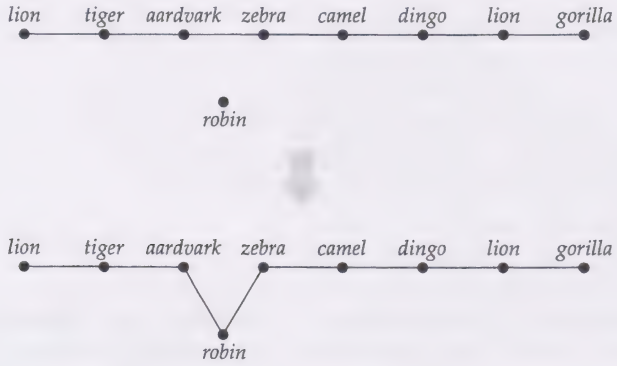
The graph is easier to draw than the computer tape, and so there is a distinct advantage in representing a stack by a path graph. Also, the vertices do not need an address attached to them as do the cells on the tape; the graph is, in a sense, address-free. For example, starting at the root vertex we can move through the graph, using the edges to go from vertex to vertex. This is not the same as the operation POP, which deletes the root vertex and its edge. Since we have deleted no vertices, we can also move back through the graph. This process is called a **graph search**.

Can this graph process be duplicated for a stack? In fact, it is not difficult to do this. All we need is a second information cell, called ITEM, that starts with the same number in it as in the cell TOP. Decreasing this number by 1 (but not allowing it to become smaller than 1), we obtain the address of the next item down the store. Increasing it by 1 (but not allowing it to get larger than the number in TOP), we obtain the address of the next item up the store. We have added a new operation to those of the stack data type — namely, a search corresponding to the graph search above.

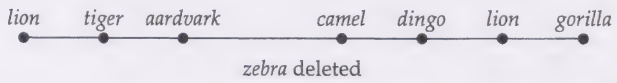
As we see in the next subsection, it is a useful exercise to represent a stack by a graph and then take the simple and obvious operations we can perform on a graph and duplicate their action for a stack.

2.2 The list data type

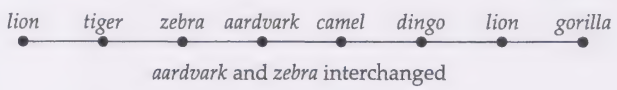
It is a simple matter to *insert* a new vertex between any two adjacent vertices in a path graph — we simply delete an edge and replace it with two edges joined to the new vertex:



It is also a simple matter to *delete* an internal vertex of a path graph and connect the graph up again — we simply delete the vertex and its incident edges and then join the vertices that were adjacent to the deleted vertex:



To *interchange* two adjacent vertices, we simply delete the first one and then insert it after the second:



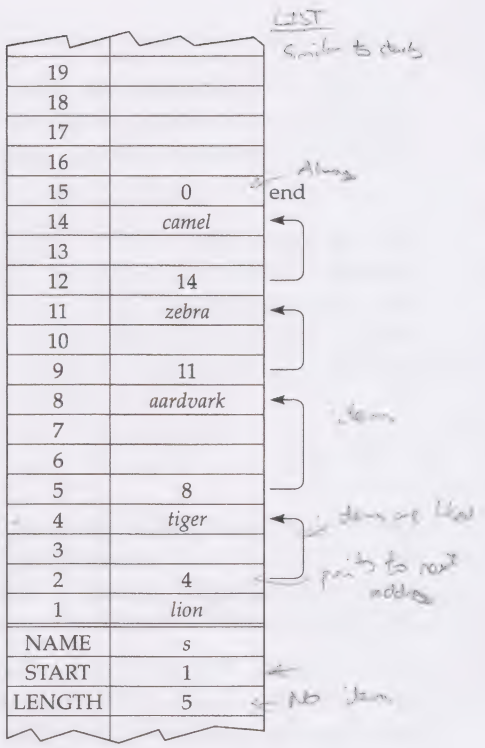
How can we duplicate these simple manipulations for data stores? Stacks are cumbersome when it comes to insertion and deletion other than at the top, since we have to keep all of the items together in a single block. For example, removing an item from the interior of the block means that we have to shift all the items above it down by one cell to preserve the block structure.

We need a way of connecting cells in different parts of the store. It is possible to move to any cell in the store if we are given its address. With this in mind, consider the tape shown in the margin. The information cell labelled START holds the address of the first item *lion*. If we go to this address and move one cell up the store, we find the address, 4, for the cell of the second item *tiger*. We say that the number in this cell *points* to the next address. We now repeat this process for the whole set of items. We say that the items are *linked*, each item having the address for the next item in the cell immediately above it. The last item in the list has the number 0 in its 'forwarding address' cell. We also have a cell labelled LENGTH that contains the number of items, and one labelled NAME that contains the name of the set of items.

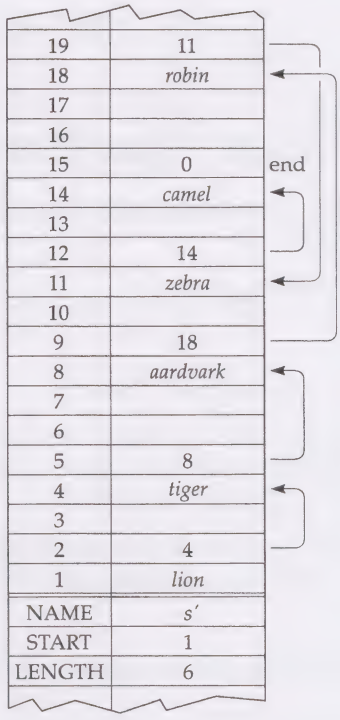
We call this data store a **list**. Lists are not very different from stacks. The data is entered in a linear fashion, one item after another, but it is not stored in adjacent cells on the tape. This makes it easy to insert an item into the list, and insertion is the main operation of this data type.

Consider the tape in the margin above. Suppose that we wish to insert the item *robin* after the third item from the start of the list, between the items *aardvark* and *zebra*. To do this, we pick up the START address and move to the first item, then to the second item, and then to the third. We store this third item's forwarding address (as the only item in a temporary stack, say) and in its place we write the address of the first of any pair of unused cells. In the cell at this address we write *robin*, and in the cell immediately above it we write the stored forwarding address (obtained from the top of the temporary stack) of the cell that contains *zebra*. We then increase the LENGTH by 1.

The *list data type* has a set of basic functions that differ slightly from those of stacks, but allow us to do more in the way of manipulating our data.



19	
18	
17	
16	
15	0
14	*camel*
13	
12	14
11	*zebra*
10	
9	11
8	*aardvark*
7	
6	
5	8
4	*tiger*
3	
2	4
1	*lion*
NAME	*s*
START	1
LENGTH	5



19	11
18	*robin*
17	
16	
15	0
14	*camel*
13	
12	14
11	*zebra*
10	
9	18
8	*aardvark*
7	
6	
5	8
4	*tiger*
3	
2	4
1	*lion*
NAME	*s'*
START	1
LENGTH	6

The operation INSERT(item, 1, k) places the item at the start of the new list.

* In the case of our tape, the address is in the cell immediately above.

Definition

The **list data type** is a data store k in which each item keeps an address* that **points** to the next item, together with the following basic operations:

FIRST(k) = the first item in the list k;

ITEM(i, k) = the ith item in the list k;

LENGTH(k) = the number of items in the list k;

INSERT(item, i, k) = insert the item after the (i–1)th item in the list k, so that it becomes the ith item of the new list formed;

DELETE(i, k) = remove the ith item from the list k.

Problem 2.3

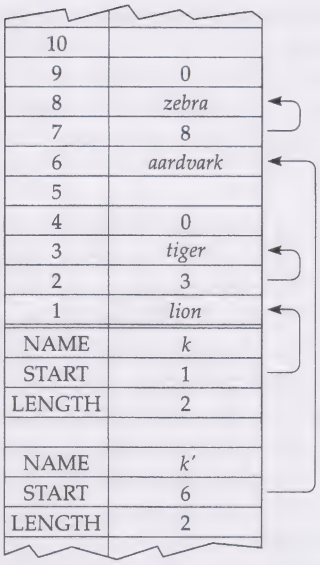
The basic operation LENGTH for a list corresponds to the basic operation DEPTH for a stack. Which basic operations for a list correspond to TOP, POP and PUSH for a stack?

It is straightforward to write an algorithm that inserts more than one item — for instance, we might want to insert another list at some point in a given list.

Note that a list requires roughly twice the storage space of an equivalent stack. To store n items of data, we require $2n$ cells plus three information cells, so the space complexity function is $S(n) = 2n + 3$. This is of order $O(n)$, which is the same as that for stacks.

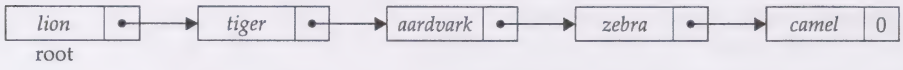
Problem 2.4

Two lists can be stored on the same tape as illustrated in the margin. Explain how the list $k' = aardvark, zebra$ can be inserted into the list $k = lion, tiger$ between *lion* and *tiger* to create a new list k''.



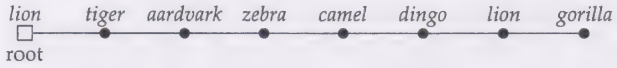
A stack is just the right type of store for this purpose: as we move on from an item in the list, we store that item's address at the top of the stack. The top item of the stack is then always the address of the previous item, and to move back along the list we just POP the addresses off the stack.

We can represent a list of length n by a rooted path digraph with n vertices. To emphasize the structure of a list, we draw the vertices and arcs as shown below. Each vertex is drawn as a pair of cells, the first of which is labelled with the item's name and the second of which points to the next item in the list. The root vertex corresponds to the *first* item in the list.



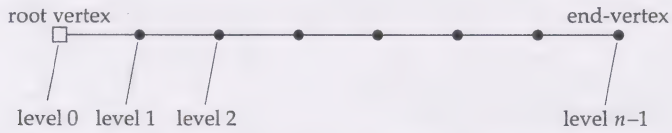
The arcs in the digraph illustrate that, in a list, we know the address of the next item in a list, but not the address of the previous item. Once we move along the list, we do not lose the list, but we have no means of backtracking unless we store the addresses somehow.

A list together with some sort of backtracking store can be represented by a rooted path graph. This representation is the same as that for a corresponding stack except that the root vertex now corresponds to the *first* item.



2.3 List algorithms

Consider a list represented by a rooted path graph P_n with n vertices and $n-1$ edges.



The vertices are all considered to be at different **level**, the root at level 0 (the *top* level), the next vertex at level 1, and so on, to the end-vertex at level $n-1$ (the *bottom* level).

We shall use the above graph representation to help us explore some search algorithms for lists, and determine their time complexity functions.

The following discussion applies equally well to a stack, and each algorithm can be written with the list operations replaced by the equivalent stack operations.

Searching for a given item

The obvious way to search a rooted path graph is to start at the root vertex and move from level to level, comparing the item at each vertex, as we reach it, with the given item. This is called **depth-first search**. For this simple search, the worst that can happen is that the item is not found in the graph, and therefore the time complexity function is $T(n) = n$, since n comparisons are made, each taking one unit of time.

Theorem 2.1

The time complexity function for a depth-first search of a rooted path graph is of order $O(n)$.

It is easy to describe the search in words when a list is considered as a graph. To describe the corresponding method of searching a data store using the operations of a list data type, we need a more formal approach. We construct an algorithm to search a list k as follows.

Algorithm: SEARCH 1 for a given item in a list k of length n

STEP 1 Set the value of a variable $i = 0$.

STEP 2 Increase the value of i by 1 and compare ITEM(i, k) with the given item. If they are the same, or if they are not the same but $i = n$, STOP. Otherwise repeat Step 2

This is a formal way of describing how we search the rooted path graph by moving along one level at a time. The first time that Step 2 is reached, $i = 1$ and so ITEM(1, k) is compared with the given item. In the worst case, when the given item is not in the list, Step 2 is carried out n times in all, until all the items in the list have been compared with the given item. Hence, as above, since a unit of time is taken for each comparison, $T(n) = n$.

The above algorithm is an *iterative* or 'looping' algorithm, as it performs Step 2 and then loops round and performs it again. There is another type of algorithm used for this type of search. It is a *recursive* algorithm, and the underlying idea is as follows. To search a list of length n, we compare the given item with the first item, and if it is not the same we discard that first item and carry out a fresh search on the resulting list of length $n - 1$.

Most algorithms in this course are presented in iterative form in the text, but generally the equivalent recursive forms are implemented on the computer packages.

Algorithm: SEARCH 2 for a given item in a list k

STEP 1 If LENGTH(k) = 0, the list is empty, so STOP.

STEP 2 Compare the given item with ITEM(1, k). If they are the same, STOP.

STEP 3 Use SEARCH 2 on the list DELETE(1, k).

How do we determine the time complexity function $T(n)$ in this case? For any list of length 0, we have $T(n) = 0$. For a list of length $n > 0$, we compare the given item with the first item in the list. This takes one unit of computing time. The next step is to DELETE the first item and search this new list. If the time taken to search a list of length n is $T(n)$, then the time taken to search this new list of length $n - 1$ is $T(n - 1)$. It follows that the function T is described by a system of equations known as a **recurrence system**:

$$T(0) = 0 \qquad (2.1)$$

$$T(n) = 1 + T(n-1) \qquad (n > 0) \qquad (2.2)$$

A recurrence system is a perfectly good way of defining a function $T(n)$ for all n. In order to evaluate $T(n)$, we use equation 2.2 several times, until eventually we can use equation 2.1. For example, we determine $T(3)$ as follows:

$$\begin{aligned} T(3) &= 1 + T(2) && \text{using 2.2} \\ &= 1 + (1 + T(1)) && \text{using 2.2} \\ &= 1 + (1 + (1 + T(0))) && \text{using 2.2} \\ &= 1 + (1 + (1 + 0)) && \text{using 2.1} \\ &= 3. \end{aligned}$$

In the same manner, we can derive a formula for $T(n)$. From the system of recurrence equations, we have:

$$\begin{aligned} T(n) &= 1 + T(n-1) && \text{using 2.2} \\ &= 1 + 1 + T(n-2) && \text{using 2.2} \\ &= 1 + 1 + 1 + T(n-3) && \text{using 2.2} \\ &\vdots \\ &= 1 + 1 + 1 + \ldots + 1 + T(1) && \text{using 2.2} \\ &= 1 + 1 + 1 + \ldots + 1 + 1 + T(0) && \text{using 2.2} \\ &= \underbrace{1 + 1 + 1 + \ldots + 1 + 1}_{n \text{ terms}} + 0 && \text{using 2.1} \\ &= n. \end{aligned}$$

We call this a **solution by iteration**.

As we would expect, since both algorithms are essentially the same, each has time complexity function $T(n) = n$, of order $O(n)$.

Problem 2.5

Suppose that the time complexity function of a certain algorithm for a list of length n is given by the recurrence system:

$T(0) = 1$

$T(n) = 2 + T(n-1) \qquad (n > 0)$

(a) Use the recurrence system to evaluate $T(3)$.

(b) Use the system to obtain a solution by iteration for $T(n)$.

Searching for repeated items

Another important operation often used on data is to search for all occurrences of repeated items in the data.

This operation can also be used to delete any repeated items.

The picture of a list as a rooted path graph gives us an idea of what is required for such a search. We start at the root vertex and compare the item at this vertex with each of the items at the other $n-1$ vertices below it, marking each vertex at which the root item occurs. Then we move to the next unmarked vertex down and compare this with the (at most) $n-2$ unmarked vertices below it. We keep moving down the levels of the graph, each time comparing the vertex we reach with all those unmarked vertices below it. This locates all occurrences of repeated items.

The time complexity function is calculated as follows. The first vertex (the root) takes $n-1$ units of time, the second vertex takes at most $n-2$ units, and so on, to the $(n-1)$th vertex, which takes at most 1 unit of time, and the nth vertex, which does not need to be compared with any other. Adding all these units of time, we have:

$$\begin{aligned} T(n) &= (n-1) + (n-2) + \ldots + 2 + 1 + 0 \\ &= n(n-1)/2 \\ &= (n^2 - n)/2. \end{aligned}$$

The sum of the *arithmetic series* $a + (a+d) + (a+2d) + \ldots + (a+(n-1)d)$ is $n(2a + (n-1)d)/2$.

We therefore have the following result.

Theorem 2.2

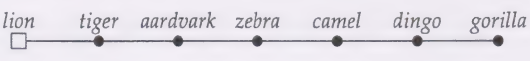
The time complexity function for a search for all repeated items in a rooted path graph is of order $O(n^2)$.

As we would expect from the relative orders of their time complexity functions, this algorithm takes longer to run than the depth-first searches for a single item. For example, for a list of 100 items, this algorithm takes almost 5000 time units compared with 100 for the depth-first searches.

The 'search for item' and the 'search for repeats' algorithms of this subsection are useful utilities, but the most important algorithms, for large amounts of data, are those that sort and order the data. Two of these are discussed in the next subsection.

2.4 Sorting algorithms

The list graph below is that of our usual example, except that we have deleted the second occurrence of the item *lion*.



Consider the problem of arranging the items in the list in lexicographic order, as in a dictionary, with the 'smallest' item *aardvark* (the first item in dictionary order) at the root vertex. Sorting is one of the major problems to engage the minds of computer scientists, as huge amounts of computer time are spent on sorting vast sets of data. There are a number of sorting methods whose names have become legendary in the history of the subject, and these methods are constantly being refined. Two of the most common are called *bubble sort* and *merge sort*. We take a brief look at each of these.

Bubble sort algorithm

This sorting algorithm takes the form of a number of 'passes' over all the items in the list. For the first pass, we start at the first item in the list, at the root vertex, and compare it with the item at the second vertex. If the first item is 'larger' (that is, it follows the second item alphabetically), then we exchange the two vertices; if the first item is not larger, then we do nothing. Next, we repeat the process for the second and third vertices, as they now are. We work our way along the graph, comparing the ith vertex with the $(i+1)$th vertex, for each i, and exchanging them if necessary; we finish the first pass with a comparison of the $(n-1)$th and nth vertices, where n is the total number of vertices. We have then made $n-1$ comparisons, and some vertices have been 'bubbled' along until they meet the first vertex larger than themselves. In particular, the largest item will have been 'bubbled' along the graph until it reaches the end-vertex, and is in its correct place in a sorted list.

The algorithm now takes a second pass along the graph, starting at the root vertex and repeating the process in exactly the same way. But now we need make only $n-2$ comparisons, the last being between the $(n-2)$th and $(n-1)$th vertices, since the largest item is already in its correct place. At the end of this pass the second largest item is in its correct place, just before the largest.

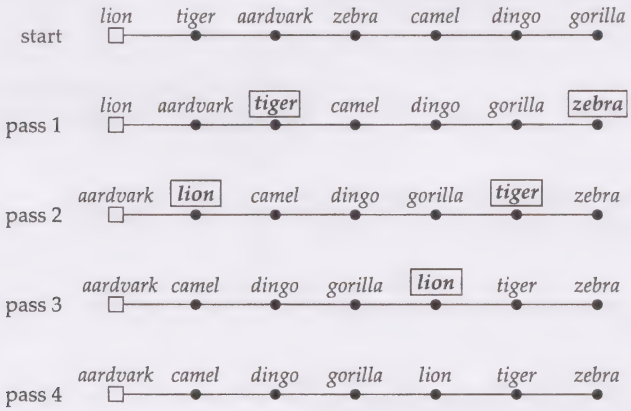
We make $n-1$ such passes in all, each pass using one fewer comparison than the previous one.

Note that we do not need n passes, since when the second smallest item is in place, the smallest is also in its correct place, at the root vertex.

As we have seen, this algorithm is easily described in terms of sorting the vertices of a graph, and nothing is gained by describing it more formally. Indeed, this is quite difficult to do in general, and does not help us when applying it to a particular example.

Example 2.1

The bubble sort algorithm takes four passes to sort our list of animals into dictionary order, as illustrated below. At each pass, the highlighted vertices are those that have been bubbled along on that pass.

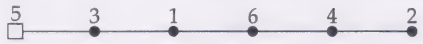


No vertices are interchanged on the fourth pass, and so the sort is completed after three passes. The list is now sorted, with *aardvark* at the root vertex and *zebra* at the end-vertex. ■

In general, the algorithm stops once no vertices are interchanged at a particular pass.

Problem 2.6

Carry out a bubble sort on the following list:



What is the order of the time complexity function for this algorithm? As before, each comparison costs 1 time unit, but an interchange, which uses the basic operations for a list, comes free. Since we make $n-1$ comparisons on the first pass, $n-2$ comparisons on the second pass, and so on,

$$T(n) = (n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2.$$

Note that the worst case, where all $n(n-1)/2$ comparisons are needed, occurs when the original list is in exactly the opposite order to that required.

Thus, the time complexity function is of order $O(n^2)$.

Theorem 2.3

The time complexity function for a bubble sort is of order $O(n^2)$.

In recursive terms, if the list has 1 item, then no comparisons are needed to sort it and $T(1) = 0$. For a list of length $n > 1$, the first pass bubbles the largest item along to the end-vertex after $n-1$ comparisons (time units). The $n-1$ items smaller than it still need to be sorted, and we do this by applying the algorithm again to a list of length $n-1$. So $T(n) = (n-1) + T(n-1)$. This gives the recurrence system:

$$T(1) = 0$$

$$T(n) = (n-1) + T(n-1) \qquad (n > 1)$$

To solve this system, we apply the second of the equations repeatedly until we can use the first equation:

$$\begin{aligned} T(n) &= (n-1) + T(n-1) \\ &= (n-1) + (n-2) + T(n-2) \\ &= (n-1) + (n-2) + (n-3) + T(n-3) \\ &\vdots \\ &= (n-1) + (n-2) + (n-3) + \ldots + 1 + T(1) \\ &= (n-1) + (n-2) + (n-3) + \ldots + 1 + 0 \\ &= n(n-1)/2. \end{aligned}$$

Merge sort algorithm

Before we can discuss this second method of sorting a list, we need a procedure for merging two *sorted* lists. Two sorted lists, of lengths m and n, can be merged into one sorted list as follows.

Algorithm: to merge two sorted lists

This algorithm 'shuffles' the two lists together, keeping the items in order.

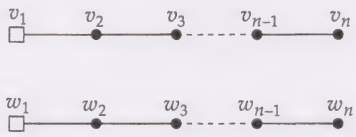
STEP 1 Start a new list, with no items in it.

STEP 2 Compare the first items in both lists, delete the smaller of these from its list, and insert it at the end of the new list. Repeat until one list is empty

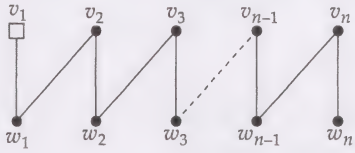
STEP 3 Insert what remains of the non-empty list at the end of the new list, and STOP.

Each comparison of an item from one list with one from the other results in an item being removed from one list and added to the new list, and so there are at most $m + n$ comparisons. In fact, since no comparison can be made when one list is empty, there are at most $m + n - 1$ comparisons. The worst

case occurs when both lists have the same length n and when, for lists with vertices v_i and w_i, respectively, we have $v_i < w_i < v_{i+1} < w_{i+1}$ for $1 < i < n - 1$.



Then the merged list is given by the following graph:



Here, all $2n - 1$ comparisons have been made, and so $T(n) = 2n - 1$. Thus, the time complexity function is of order $O(n)$.

Theorem 2.4

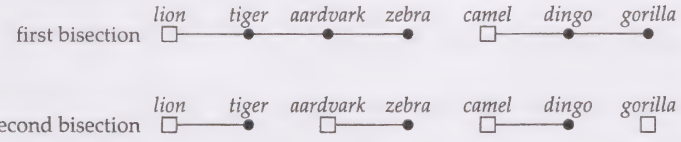
The time complexity function for merging two sorted lists of lengths m and n, where $m \leq n$, is of order $O(n)$.

We now return to the main problem, that of sorting a list of length n. The merge sort makes use of the above algorithm in the following way. We first bisect the list into two lists of equal (or nearly equal) length. We then bisect each such half-length list into two equal (or nearly equal) quarter-length lists. The bisection process continues until we obtain a number of pairs of short lists, each with one or two items. These short lists are easily sorted, as there is nothing to do for a one-item list, and a single comparison to make for a two-item list. The pairs of short lists are now merged, using the above algorithm, to give pairs of lists, each with three or four items. We merge these pairs, and so on, until the whole list is sorted.

Example 2.2

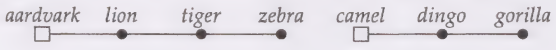
We apply the merge sort algorithm to the following list.

We first bisect the lists until we have lists of lengths 1 and 2.



We next sort these short lists; in fact, they are already sorted.

Then we merge the pairs using the merge algorithm, to get:



Finally we merge these two lists using the merge algorithm to get:

The original list is now fully sorted. ■

Problem 2.7

Carry out a merge sort on the following list:

In order to determine the time complexity function for the merge sort algorithm, we simplify the calculations by supposing that the original list has length $n = 2^k$, for some positive integer k. The first bisection gives two lists, each of length 2^{k-1}. Bisecting each of these gives four lists of length 2^{k-2}, and so on. The recurrence system for the time complexity function can now be determined as follows.

If the list has just two elements, then $T(2) = 1$, as a single comparison is needed to sort it. For a list of length 2^k (where $k > 1$), we first sort *two* lists, each of length 2^{k-1}, which takes $2 \times T(2^{k-1})$ units of time, and then merge them using the above algorithm, which takes

$$2n - 1 = 2 \times 2^{k-1} - 1 = 2^k - 1 \quad \text{units of time.}$$

If we assume, to make the calculation easier, that it takes the slightly longer time of 2^k units to merge the two lists, then the time taken to sort a list of length 2^k is

$$T(2^k) = 2T(2^{k-1}) + 2^k.$$

The resulting recurrence system is:

$$T(2) = 1$$

$$T(2^k) = 2^k + 2T(2^{k-1}) \qquad (k > 1)$$

Applying the second equation repeatedly, we obtain:

$$\begin{aligned} T(2^k) &= 2^k + 2T(2^{k-1}) \\ &= 2^k + 2(2^{k-1} + 2T(2^{k-2})) \\ &= 2^k + 2^k + 2^2T(2^{k-2}) \\ &\vdots \\ &= \underbrace{2^k + 2^k + \cdots + 2^k}_{k-1 \text{ terms}} + 2^{k-1}T(2) \\ &= 2^k\,(k - \tfrac{1}{2}), \qquad \text{since } T(2) = 1. \end{aligned}$$

Now, $n = 2^k$, and so $k = \log_2 n$; we can therefore write the above result as

$$T(n) = n(\log_2 n - \tfrac{1}{2}).$$

In fact, the following general result can be established for *any* n.

Theorem 2.5

The time complexity function for a merge sort on a list of length n is of order $O(n\log_2 n)$.

In the order hierarchy, $O(n\log_2 n) \subset O(n^2)$, and so the merge sort algorithm is quicker than the bubble sort algorithm.

More generally, it can be shown that the worst-case behaviour for merge sort is the best that we can achieve when sorting a list. In other words, no list-sorting algorithm can have a time complexity function of order smaller than $O(n\log_2 n)$.

Problem 2.8

Use the time complexity functions $T(n) = n(n-1)/2$ and $T(n) = n\log_2 n$ to compare the worst-case speeds of the bubble sort and merge sort algorithms applied to a list with 32 items.

We conclude this section by mentioning that the above formulation for merge sort is an example of a problem-solving methodology called *divide and conquer*.

Definition

The **divide-and-conquer method** of problem solving breaks down a problem into subproblems, solves these subproblems, and then combines the solutions to the subproblems into a solution for the original problem. The method is *recursive;* the subproblems themselves can be solved using the divide-and-conquer technique.

This method appears again in the next section.

After studying this section, you should be able to:

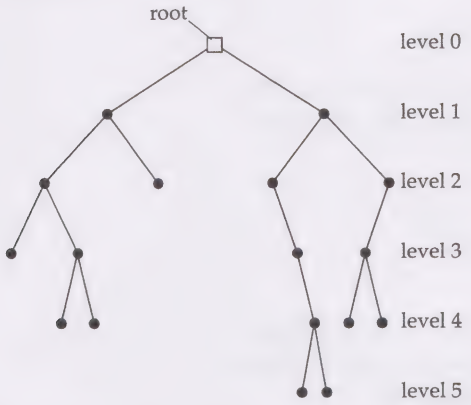
- understand what are meant by *stack data type* and *list data type;*
- represent stacks and lists graphically;
- use list-searching algorithms for finding a given item in a list;
- use *bubble sort* and *merge sort algorithms* to sort a list;
- compute time complexity functions, and their orders, for the given algorithms.

3 Binary trees

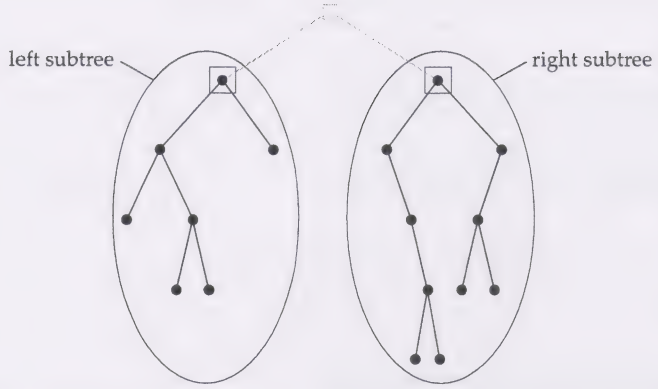
In *Graphs 2* you met the idea of a binary tree — a rooted tree in which there are at most two downward branches (one to the left and one to the right) from each vertex. In this section we study binary trees, with particular reference to search procedures.

3.1 Binary tree data type

A small binary tree is shown below.



We can 'store' data at each vertex, and with this in mind we adopt certain conventions. The root is drawn at the top of the tree (level 0) and there are at most two vertices adjacent to it and drawn at the same level below it (level 1); these vertices are ordered, in the sense that one is to the left and one to the right of the root. In turn, each of these vertices has at most two vertices below it (a left and a right), and all such vertices are drawn at the same level (level 2), and so on. Note that this ordering of left and right allows us to talk of a *left subtree* and *right subtree,* as illustrated overleaf.



For each vertex v of a binary tree, an adjacent vertex below and to the left can be regarded as the root of another binary tree, known as the *left subtree of v*, and similarly an adjacent vertex below to the right is the root of the *right subtree of v*.

Each subtree also has a binary tree structure, and each in turn has a left subtree and/or a right subtree. And so on. Thus a binary tree has a recursive structure.

This structure is of importance in storing data. Up to now, we have mainly thought of data being stored at the vertices of the graph of a list. We now consider how to change the list structure into that of a binary tree.

Growing binary trees

We can construct a binary tree from a list using the recursive method of divide and conquer, as follows.

Algorithm: GROW-TREE for constructing a binary tree from a list

STEP 1 If the list is empty, STOP.

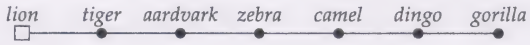
STEP 2 Find the centre or bicentre of the graph of the list. If it has a centre, then choose this as the root of the binary tree; if it has a bicentre, then choose the rightmost vertex of the two as the root.

Centres and bicentres of trees were introduced in *Graphs 2*, Section 2.3.

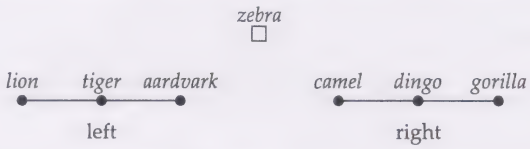
STEP 3 The root divides the list into a *left* list and a *right* list (one or both of which may be empty). Apply GROW-TREE to both of these to obtain the left and right subtrees.

Example 3.1

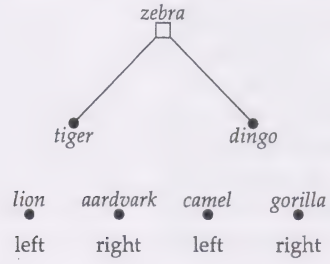
Consider the following list:



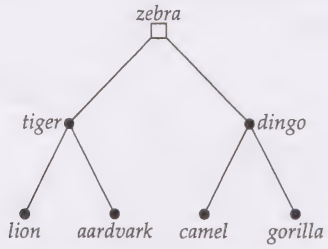
The centre of this graph is the vertex *zebra*, and so this is the root of the tree. The centre divides the graph into left and right lists as shown below.



We now apply GROW-TREE to these two lists. The roots of the left and right subtrees are the centres *tiger* and *dingo* respectively. Each of these centres divides its respective list into a left list and a right list.



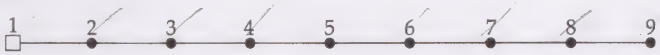
Each of these four lists has just one vertex and each of these vertices forms the root of a subtree. In all four cases, the corresponding left and right sublists are empty, and so the process stops. This gives us the following binary tree.



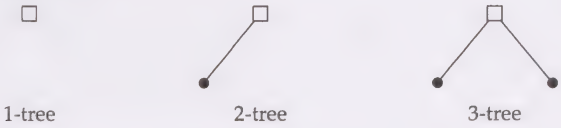
■

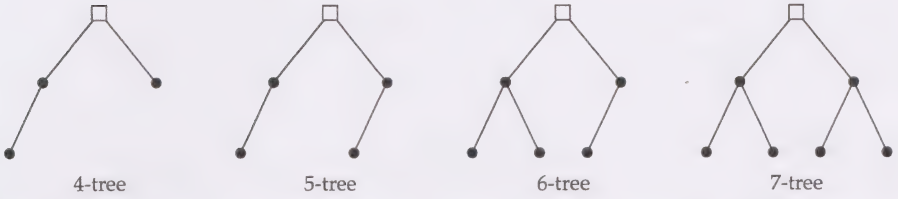
Problem 3.1

Use the GROW-TREE algorithm to construct a binary tree from the following list.



The use of the GROW-TREE algorithm on lists of lengths 1, ..., 7 yields the following catalogue of binary trees.

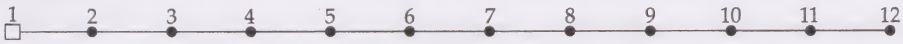




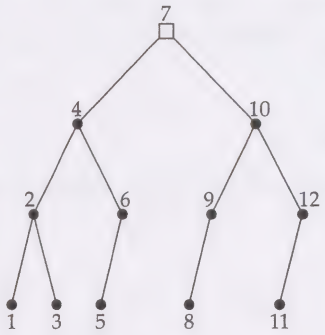
The catalogue can be extended by applying GROW-TREE to longer lists. As the number of items in the list increases, the vertices at each level are filled up from the left, first on the left branches and then on the right branches. For each value of $k = 0, 1, 2, \ldots$, the kth level has a maximum of 2^k vertices.

As well as applying GROW-TREE to longer lists, we can also grow bigger binary trees by using the above catalogue of seven trees and exploiting the recursive nature of the algorithm, as the following example shows.

Example 3.2



We shall turn this 12-item list with into a binary tree. The root is at the rightmost bicentral vertex 7, the left subgraph (vertices 1–6) has length 6, and the right subgraph (vertices 8–12) has length 5. We know that GROW-TREE applied to lists of lengths 6 and 5 gives the 6-tree and 5-tree in the catalogue above. We can therefore deduce that we need to 'hang' the 6-tree on the left edge from the root, and the 5-tree on the right edge. This gives the binary tree in the margin. ■

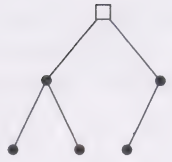


The following definition is useful when we wish to classify binary trees.

Definition

The **height** of a binary tree is the number of vertices in a path of maximum length, starting from the root.

Thus the height of a binary tree with $k + 1$ levels $(0, 1, \ldots, k)$ is $k + 1$ — that is, it is the number of levels in the tree. For example, the binary tree in the margin has height 3.



The GROW-TREE algorithm fills up each level of vertices before going on to the next level, and produces examples of *balanced* binary trees.

Definition

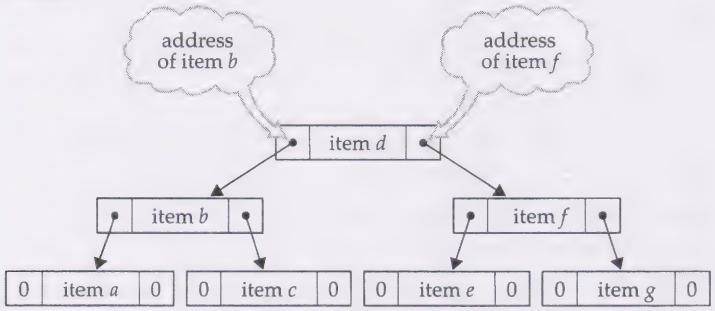
A binary tree is **balanced** if, at each vertex v, the heights of the left subtree of v and the right subtree of v do not differ by more than 1.

Problem 3.2

Are the following binary trees balanced?

Storing a binary tree on a tape

In order to store a list as a binary tree in a data store tape, we use the scheme below.



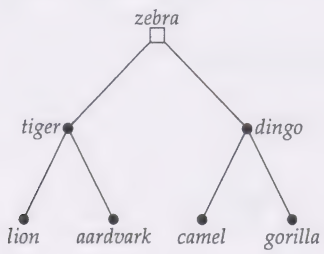
Each item is stored in a cell that has two neighbouring cells with tape addresses stored in them. One address points to the cell where the item at the root of the left subtree is stored, and the other points to the cell where the item at the root of the right subtree is stored. Each of these cells has two neighbouring cells in which to store the addresses of their left and right subtrees, and so on. The items at the end-vertices have 0 instead of an address in their neighbouring cells.

Each set of three cells (one for the item, and two for the left and right addresses) can be situated in any three neighbouring cells on the tape, while the address of the root item is stored in an information cell with the name ROOT.

The *empty tree* — the tree with no vertices — is denoted by 0 in the information cell ROOT.

Example 3.3

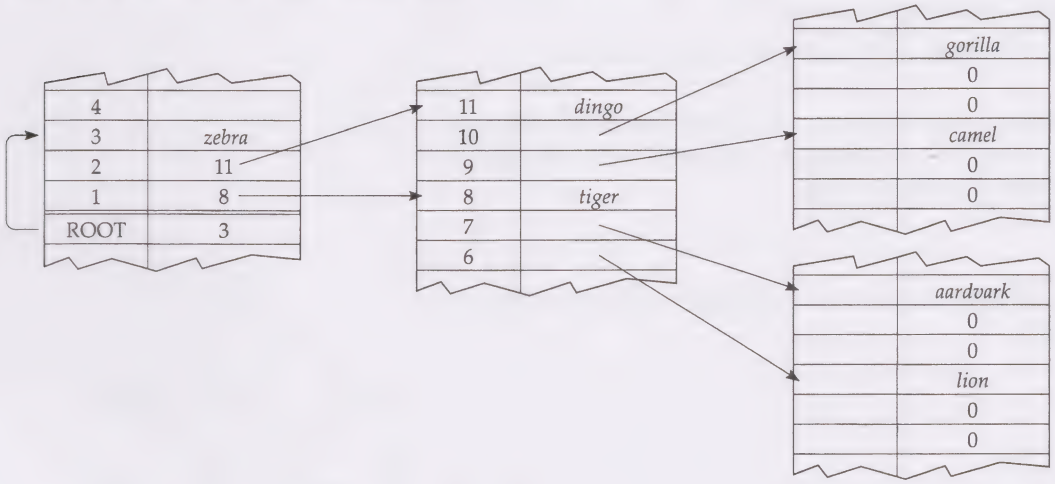
We shall store the following binary tree on a data store tape.



Sections of the tape store are shown below. We have stored the item *zebra* at the root of the tree at cell 3, and this cell's address is stored in the information cell labelled ROOT, as shown. The cells 2 and 1, immediately below cell 3, contain the addresses of the root of the right subtree (cell 11) and the root of the left subtree (cell 8), respectively.

We adopt the commonly used convention that for an item stored in cell k, cell $k-2$ points to the left subtree and cell $k-1$ points to the right subtree.

The next bit of tape shows *dingo* stored at cell 11, and *tiger* stored at cell 8. The two cells below these cells point in turn to the next level of right and left subtrees, as shown. (We have not filled in the actual addresses, as it is not important what these are.) The items stored at the end-vertices have zeros in the two cells below them.



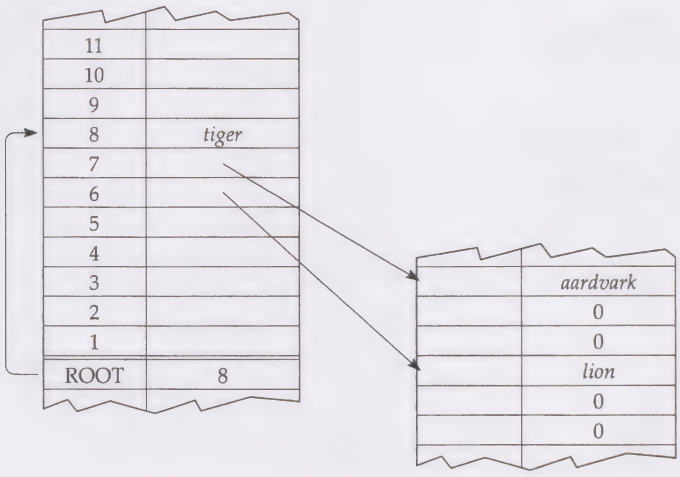
With the sections of the tape pictured in this way, you can easily see the original tree structure — laid on its side, as it were.

■

To store n items, a stack requires $n + 1$ cells of our tape model, and a list requires $2n + 3$ cells, whereas a binary tree requires $3n + 1$ cells. However, the extra space required to store a binary tree is compensated for by the ease with which data stored in this manner can be manipulated.

In other words, the space complexity function for a binary tree data store is $S(n) = 3n + 1$, of order $O(n)$.

We can easily extract the left subtree. We first find the address of the left subtree's root item — in this case, cell 8. We then place this address in the information cell ROOT, and obtain the storage tape below. (The right subtree can be extracted just as easily.)



Given two stored trees, T_1 and T_2, we can build a larger tree as follows. Store a new root item in some cell — that is, put this cell's address in the information cell ROOT. Then place the root address of T_2 in the cell below the new root item's cell, and the root address of T_1 in the cell below that. This makes T_1 the left subtree of the new tree, and T_2 its right subtree.

These simple procedures define the basic operations for the *binary tree data type*.

Definition

The **binary tree data type** is a data store T, corresponding to a binary tree, in which each item keeps two addresses that point to its two subtrees, together with the following basic operations:

ROOT(T) = the item at the root vertex;

LEFT(T) = the left subtree of T;

RIGHT(T) = the right subtree of T;

MAKE-TREE(T_1, item, T_2) = the tree constructed with the item as its root, T_1 as its left subtree and T_2 as its right subtree;

ISEMPTYTREE? = TRUE if T is the empty tree, and FALSE otherwise.

With these simple operations we can build up algorithms to manipulate data stored as a binary tree. For simplicity, we work with the graph representation of data stored in a binary tree.

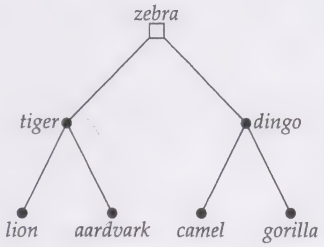
Problem 3.3

Let T be the binary tree in the margin. Using the basic operations above, determine the following:

(a) LEFT(T);

(b) ROOT(RIGHT(T));

(c) MAKE-TREE(T, *yak*, T).



3.2 Binary search trees

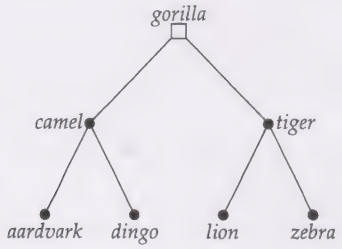
We now consider an important type of binary tree. We again start with a list, but first we order the list using any sorting algorithm from the previous section. To this ordered list we then apply the GROW-TREE algorithm as before. The resulting tree is called a *binary search tree.*

Example 3.4

We construct a binary search tree for the following list:

We first order the list:

Next we apply GROW-TREE to get the binary search tree:



■

Definition

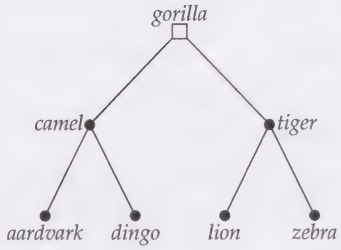
Given a set of items that can be ordered, a **binary search tree** is a binary tree in which each vertex v represents one of these items in such a way that:

- v is larger than each item below and to the left of v;
- v is smaller than each item below and to the right of v.

In a binary search tree, you know where you are in a search: at each vertex you go left for something smaller and right for something bigger.

Example 3.4 continued

We now search the binary search tree constructed above for the item *quagga*.



First we compare the root item with the search item. Since *quagga* is larger than *gorilla* (that is, after it in the dictionary), we move to the right, to the vertex *tiger*. Now the search item is smaller, so we move to the left, to *lion*. The search item is larger, but since we are at an end-vertex, we can go neither left nor right; the search is finished and the item is not found.

At worst, we need only make three comparisons in searching for any item in this tree, one for each level of the tree, as compared with seven comparisons in searching the corresponding list. ■

In a binary search tree, a search is a matter of 'falling down' the tree, one level at a time, making *one* comparison at each level. It follows that the time complexity function is equal to the height h of the tree — that is, $T(h) = h$ — and is of order $O(h)$.

In terms of the basic functions of the binary tree data type, the algorithm for searching a binary search tree is as follows.

Algorithm: SEARCH a binary search tree

STEP 1 Compare the search item with the ROOT. If they are the same, STOP.

STEP 2 If the search item is smaller than the ROOT, then apply SEARCH to the LEFT tree, if it exists, or else STOP.
If the search item is larger than the ROOT, then apply SEARCH to the RIGHT tree, if it exists, or else STOP.

This algorithm is recursive, and the recurrence system for its time complexity function is given, in terms of the height h, by:

$$T(1) = 1$$
$$T(h) = 1 + T(h-1) \qquad (h > 1)$$

The second equation simply asserts that we make one comparison and then search a tree of height $h - 1$. In the worst case, when the item is not in the tree, we drop down the whole tree, with

$$T(h) = \underbrace{1+1+\ldots+1}_{h \text{ terms}} = h \quad \text{time units taken.}$$

This confirms that the time complexity function is of order $O(h)$.

We can also compute the order of magnitude of the time complexity function in terms of the number of vertices n in a binary search tree. A binary search tree of height h has h levels $k = 0, 1, \ldots, h-1$. At the kth level, there are s_k vertices, where

$$1 \le s_k \le 2^k \qquad (k = 0, 1, \ldots, h-1).$$

On summing this inequality over all h levels, we obtain

$$\underbrace{1+1+\ldots+1}_{h \text{ terms}} \le s_0 + s_1 + \ldots + s_{h-1} \le 1 + 2 + \ldots + 2^{h-1},$$

giving

$$h \le s_0 + s_1 + \ldots + s_{h-1} \le 2^h - 1.$$

The sum of the *geometric series* $a + ar + ar^2 + \ldots + ar^{n-1}$ is $a(r^n - 1)/(r - 1)$.

The lower bound corresponds to the case where the number of vertices equals the number of levels in the tree. In this case we have essentially a list, and no advantage is gained from having a tree structure: the time complexity function is $T(n) = n$, of order $O(n)$.

The upper bound corresponds to the most balanced (worst) case, where each of the possible vertices at every level is used up (that is, each has an item

stored at it). Since the total number of vertices is $n = s_0 + s_1 + \ldots + s_{h-1}$, we have

$$n \leq 2^h - 1, \qquad \text{giving} \quad n + 1 \leq 2^h,$$

and on taking logarithms

$$\log_2(n + 1) \leq h.$$

This tells us that the height of a binary search tree must be at least $\log_2(n + 1)$ and that, in the most balanced (worst) case , we have

$$\log_2(n + 1) = h = T(h).$$

Hence, in terms of the number n of vertices, the time complexity function is

$$T(n) = \log_2(n + 1),$$

which is of order $O(\log_2 n)$.

Since $O(\log_2 n) \subset O(n)$, we should try to obtain binary search trees that are as balanced as possible. One way is to construct them using the GROW-TREE algorithm. In this case, all the vertices are used up, apart possibly from some at the lowest level. Therefore, we have

$$2^{h-1} - 1 < n \leq 2^h - 1, \qquad \text{giving} \quad 2^{h-1} < n + 1 \leq 2^h.$$

Taking logarithms, we get

$$h - 1 < \log_2(n + 1) \leq h,$$

so that

$$\log_2(n + 1) \leq h < \log_2(n + 1) + 1.$$

Hence, the time complexity function for SEARCH, which we know to be of order $O(h)$, must be of order $O(\log_2 n)$ in this case — since h is bounded by two functions of order $O(\log_2 n)$.

This last result can be generalized to any balanced binary search tree.

Theorem 3.1

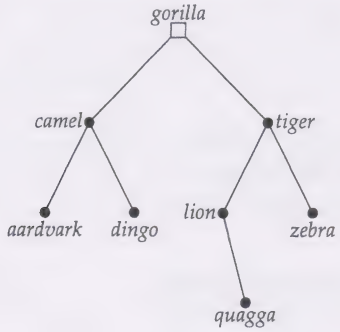
The time complexity function for a search on a binary search tree of height h is of order $O(h)$.

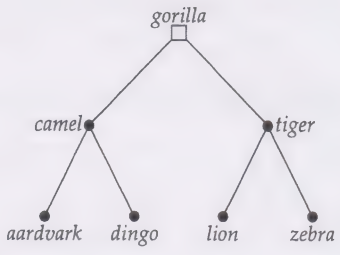
The time complexity function for a search on a balanced binary search tree with n vertices is of order $O(\log_2 n)$.

The power of a binary search tree as type of data store can be deduced from the fact that a search of a balanced binary search tree of height 20 and with data stored at each of its vertices, including all $2^{20-1} = 2^{19} = 524\,288$ end-vertices, requires at most 20 comparisons.

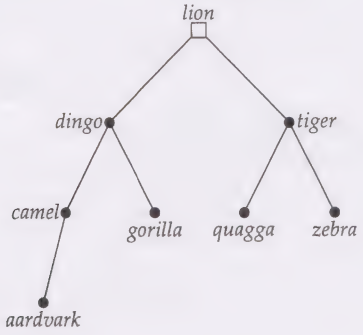
Example 3.4 continued

When we searched the binary search tree in the margin for the item *quagga* we failed to find it. At the end of the search, however, we were in a position to insert the missing item into the tree while maintaining the structure of the tree. Since we finished at the end-vertex *lion* and the given item is larger, we add it below and to the right of that vertex, as follows:





Notice that this insertion does not result in the same binary search tree (shown below) as we would get by first inserting *quagga* into its correct position in the sorted list and then growing a binary search tree from this list.



■

If we keep on inserting items into a binary search tree in this fashion, the tree may quickly become unbalanced, as the following problem illustrates.

Problem 3.4

Consider the binary search tree of Example 3.4.

(a) Search for, and then insert, the items *quagga* (as above) and *rhino*. Is the resulting binary tree balanced?

(b) Compare the tree you constructed in part (a) with the one obtained by first inserting the two new items into the ordered list and then applying GROW-TREE to the result.

To maintain their efficiency as data stores suitable for searching, we should not insert more than a small number of new items into a previously constructed binary search tree. Binary search trees into which many new items need to be inserted should be reconstructed from a new ordered list, in order to maintain their balance.

3.3 Depth-first and breadth-first searches

In this subsection, we consider methods for searching a *general* binary tree, which is not necessarily a binary search tree. There are two well-known search methods for general binary trees. They are usually known as *depth-first search* and *breadth-first search*. Each method lists the vertices as they are encountered, and indicates the direction in which each edge is first traversed. The methods differ only in the way in which the vertex lists are constructed.

You met depth-first-search in the context of lists in Section 2.3.

Although the methods are introduced in the context of binary trees, we also indicate how they can be applied to more general types of connected graph. In such applications, they effectively search the graph by searching through all the vertices of an appropriate spanning tree.

Depth-first search on a binary tree

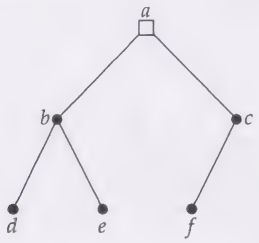
The basic idea of a depth-first search is to penetrate as deeply as possible down a binary tree before fanning out to the other vertices. Our aim is to visit every vertex, and a search for a particular item stored at a vertex could take place as each vertex is visited.

We start at the root, and visit the vertices by moving down a level each time until we reach an end-vertex. We shall be systematic about the search, going *left* whenever we can, and right otherwise. When we reach an end-vertex, we backtrack to the adjacent vertex above it and, if possible, go down to a right adjacent vertex. If this is not possible, then we

backtrack one more vertex and try again. The process is repeated, always going left to a new vertex, and backtracking to go right. Eventually, all of the vertices of the left subtree will be visited, and then we backtrack to the root and visit all the vertices in the right subtree. We finally arrive back at the root, and the search is complete.

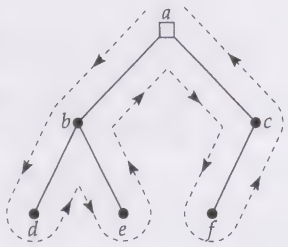
Example 3.5

Using the above procedure, we perform a depth-first search on the following binary tree, to produce a list of vertices in the order in which they are *first* visited.



Each time we meet a vertex for the first time, we write it in bold type. We start at vertex ***a*** and go left to vertex ***b***. This is not an end-vertex, so we go left to vertex ***d***. This is an end-vertex, so we backtrack to *b* and now we can go down right to vertex ***e***. This is an end-vertex, so we backtrack to *b*. Since we have already visited the left and right subtrees of *b*, we backtrack to the root *a*. All the vertices of the left subtree have now been visited. We now go right to vertex ***c***. This is not an end-vertex, so we go left to vertex ***f***. We backtrack to *c*, and cannot go right, so we backtrack to *a*, and the search is complete. The order in which we visited the vertices is given by the following list:

■



Note that if we apply GROW-TREE to this list, we do not obtain the binary tree with which we started.

The symmetry of the search described above — first go left, then go right — leads naturally to a recursive algorithm.

Algorithm: DEPTH-FIRST SEARCH on a binary tree

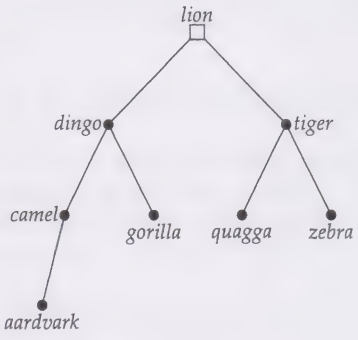
STEP 1 Start at the ROOT. Add it as the rightmost vertex of the list. If there are no subtrees, STOP.

STEP 2 If there is a LEFT subtree, then apply DEPTH-FIRST SEARCH to the LEFT subtree.
If there is a RIGHT subtree, then apply DEPTH-FIRST SEARCH to the RIGHT subtree.

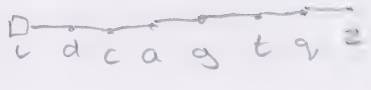
This recursion algorithm has its own built-in backtracking procedure. At any root with both left and right subtrees, it first does a depth-first search on the left subtree and then 'remembers' to go back to this root to do a depth-first search on the right subtree.

Problem 3.5

Perform a depth-first search on the binary tree in the margin, and write down the resulting list.



Recall that, in a list or a binary tree stored on a tape, we have the addresses of the vertices to go to *next*, as we move along or down the structure, but we have no addresses of the vertices that we have just come from. In order to be able to perform the backtracking required by a depth-first search, we need to be able to keep track of these, and the best structure for doing this is a *stack*. We push the vertices onto a stack as we meet them. When we backtrack, we pop the top vertex off the stack and the new top vertex is then the vertex to which we backtrack.



Example 3.5 continued

As we perform a depth-first search on the tree of Example 3.5, we keep track of the vertices visited by pushing them onto a stack. We draw the stack on its side, so that the top item is the rightmost item. The procedure is shown in full in the following table.

We did this earlier when we represented a stack by a graph, but for reasons of space we omit the graphs here.

movement on tree	stack operation	stack of vertices	current vertex
–	–	empty	–
start	PUSH ***a***	*a*	*a*
down left	PUSH ***b***	*ab*	*b*
down left	PUSH ***d***	*abd*	*d*
backtrack	POP	*ab*	*b*
down right	PUSH ***e***	*abe*	*e*
backtrack	POP	*ab*	*b*
backtrack	POP	*a*	*a*
down right	PUSH ***c***	*ac*	*c*
down left	PUSH ***f***	*acf*	*f*
backtrack	POP	*ac*	*c*
backtrack	POP	*a*	*a*
backtrack	POP	empty	–

To record the list of the items as they are first visited, we simply record the vertex each time a PUSH is made. This gives the same list as before:

■

For a binary tree with n vertices, we start with just the root vertex in the stack. Since each vertex pushed onto, or popped off, the stack represents moving along an edge of the graph, and since we move along each edge exactly twice, once on the way down and again as we backtrack, there are exactly $2(n-1)$ changes to the stack before the search is completed. For the above example with 6 vertices, the stack starts with the single vertex a, and then there are $2(6-1) = 10$ changes to the stack, given by

Recall that a tree with n vertices has $n-1$ edges.

ab, abd, ab, abe, ab, a, ac, acf, ac, a.

Note that, in a depth-first search, the depth of the stack never exceeds the height of the binary tree.

Depth-first search on a rooted tree

The stack described above becomes a very useful tool if we drop the 'first left, then right' condition for a depth-first search. Suppose that we require only that, as each vertex is reached for the first time, the search does not backtrack from this vertex until all of the edges leading down from it have been explored. In other words, it does not matter whether we go first down left and then right, as long as we do both before backtracking. This can be effected by the following algorithm.

Algorithm: for manipulating the stack in a depth-first search

STEP 1 PUSH the root vertex onto the empty stack.

STEP 2 If the top vertex of the stack is adjacent to a new vertex, PUSH this vertex onto the stack. Otherwise, POP the top vertex off the stack.

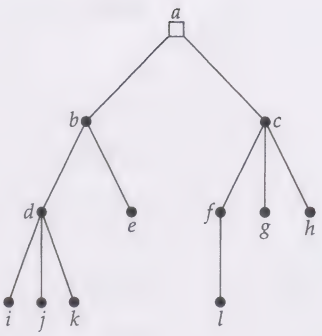
A *new* vertex is one that is not currently and has not previously been in the stack.

STEP 3 If the stack is empty, STOP. Otherwise, return to Step 2.

This algorithm can be applied to any rooted tree, not just to a binary tree, as there is no reference to left or right. Note that there is now more than one way in which the search can be made, and so the algorithm can produce more than one list for a depth-first search.

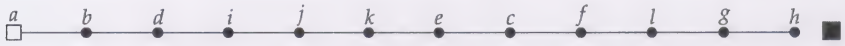
Example 3.6

Consider the rooted tree shown. This is not a binary tree, but we can still elect to search from left to right at each level. Using the above algorithm, we evolve the stack as follows:



a, ***ab***, *abd*, *abdi*, *abd*, *abdj*, *abd*, *abdk*, *abd*, *ab*, *abe*, *ab*, *a*, *ac*, *acf*, *acfl*, *acf*, *ac*, *acg*, *ac*, *ach*, *ac*, *a*.

The list of vertices in the order in which they are first visited is:



■

Problem 3.6

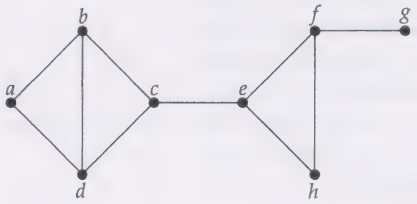
Produce a different depth-first search of the tree in Example 3.6, by working from right to left at each level.

Depth-first search on a connected graph

The algorithm for manipulating the stack in a depth-first search can be used to search any connected graph, as it makes no reference to the structure being a tree. The only change is that the 'root vertex' in Step 1 becomes the 'starting vertex' — any vertex can be chosen as the starting vertex.

Example 3.7

Consider the following graph.



We can perform a depth-first search on this graph by starting at *a*, say, going to a new vertex *b*, then *c* and then *d*. At this point there is no new adjacent vertex, so we backtrack to *c*, go to *e*, then *f* and *g*, backtrack to *f*, go to *h*, and backtrack to *a*. This completes the search.

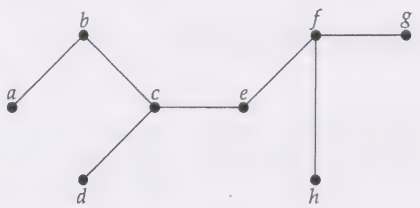
Other searches are possible.

The algorithm manipulates the stack as follows:

a, ***ab***, *abc*, *abcd*, *abc*, *abce*, *abcef*, *abcefg*, *abcef*, *abcefh*, *abcef*, *abce*, *abc*, *ab*, *a*.

The list of vertices in the order in which they are visited is:

We can also view the result in terms of the vertices visited, together with the edges actually traversed, both forwards and in backtracking. This yields the following spanning tree.



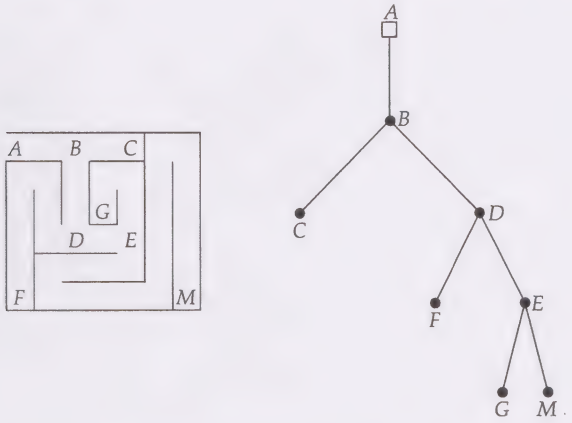
Such a tree is called a **depth-first search spanning tree**. ■

A depth-first search is really a powerful technique for determining a spanning tree in a connected graph. Some of its applications are in finding a method for visiting the various rooms of the adventure games that are so popular in the recreational use of computers. The search always proceeds along a passage to an unvisited room, as long as this is possible, and then backtracks to a point where the search can proceed as before.

Example 3.8

Consider the problem of Theseus and the minotaur. Theseus had to slay the minotaur at the centre of a labyrinth, and to ensure that he could find his way out of the labyrinth he unwound a ball of string as he made his way to the centre. The labyrinth can be represented by a graph, and the ball of string plays the part of the stack in the algorithm for a depth-first search of the graph.

You saw how mazes can be represented by graphs in the *Introduction* unit.



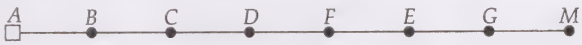
Suppose that the labyrinth and its graph are as shown, and that Theseus starts at A and has to visit the rooms C, G and F in order to find vital pieces of equipment to help him to slay the minotaur at M. After all this excitement, he must then find his way out of the labyrinth.

A depth-first search on this binary tree gives rise to the following stack, which represents the path of Theseus's ball of string:

slaying the minotaur: ***A***, ***AB***, *ABC*, *AB*, *ABD*, *ABDF*, *ABD*, *ABDE*, *ABDEG*, *ABDE*, *ABDEM*;

escaping from the maze: *ABDE*, *ABD*, *AB*, *A*.

The list of rooms and junctions in the order in which they are visited is:



■

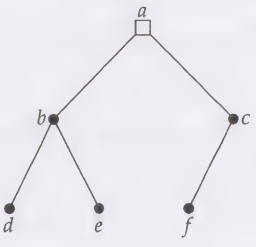
Breadth-first search on a binary tree

The basic idea of breadth-first search is to fan out to as many vertices as possible before penetrating deep into the graph. This means that we must visit all the vertices adjacent to the current vertex before going on to another vertex. For a binary tree, we visit each vertex on a particular level before repeating the process at the next level. Again, we adopt the convention that we visit the vertices from left to right at each level.

Example 3.9

Consider again the tree of Example 3.5. From the diagram, the search is very easy to write down, working from left to right at each level. The list of the vertices visited in a breadth-first search is:

This search starts with the root vertex a at level 0. We must then visit each vertex adjacent to a — vertices b and c at level 1 — before we are

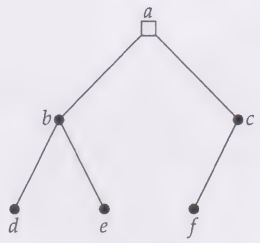


finished with *a*. You can think of the vertex *a* as entering a 'queue' and then leaving it after being 'served'. We must then visit each vertex adjacent to *b*, then those adjacent to *c*, and so on.

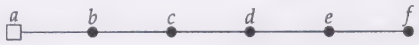
A store known as a **queue** is useful to keep track of this procedure. We INSERT vertices at the end of the queue as they are first visited, and DELETE them from the front of the queue as they are served. The vertex at the front of the queue is thus always the first to leave the queue.

A *queue* is a data store similar to a *stack*; the difference is that a queue is FIFO (First In, First Out) whereas a stack is LIFO (Last In, First Out).

movement	list operation	queue	vertex being served	still adjacent
–	–	empty	–	–
level 0	INSERT ***a***	*a*	*a*	*b, c*
level 1	INSERT ***b***	*ab*	*a*	*c*
	INSERT ***c***	*abc*	*a*	–
a served	DELETE	*bc*	*b*	*d, e*
level 2	INSERT ***d***	*bcd*	*b*	*e*
	INSERT ***e***	*bcde*	*b*	–
b served	DELETE	*cde*	*c*	*f*
	INSERT ***f***	*cdef*	*c*	–
c served	DELETE	*def*	*d*	–
d served	DELETE	*ef*	*e*	–
e served	DELETE	*f*	*f*	–
f served	DELETE	empty	–	–



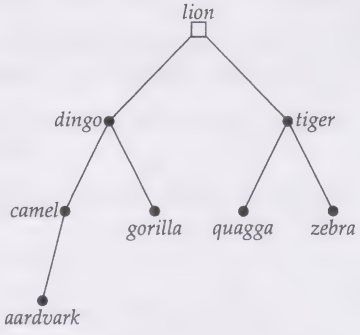
To record the list of items as they are first visited, simply record the vertex each time an INSERT is made, to get the same list as before:



■

Problem 3.7

Perform a breadth-first search on the binary tree in the margin, and write down the resulting list.



The major difference between a breadth-first search and a depth-first search is that we use a queue (FIFO) to record the former and a stack (LIFO) to record the latter. The algorithm for manipulating the queue in a breadth-first search, corresponding to that for manipulating the stack in a depth-first search, is as follows.

Algorithm: for manipulating the queue in a breadth-first search

STEP 1 INSERT the root vertex into the empty queue.

STEP 2 If the first vertex of the queue is adjacent to a new vertex, INSERT this vertex at the end of the queue. Otherwise, DELETE the first vertex of the queue.

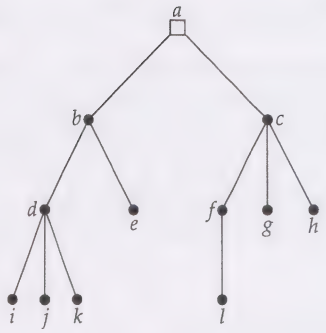
STEP 3 If the queue is empty, STOP. Otherwise, return to Step 2.

Breadth-first search on a rooted tree

As for a depth-first search, a breadth-first search can be applied to any rooted tree, not just a binary tree.

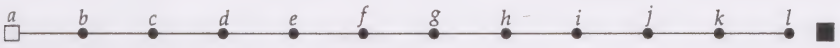
Example 3.10

Consider the rooted tree shown. This is not a binary tree, but we can still elect to search from left to right at each level. Using the above algorithm, we build up the queue as follows:



a, ***ab***, *abc*, *bc*, *bc**d***, *bcd**e***, *cde*, *cde**f***, *cdef**g***, *cdefg**h***, *defgh*, *defgh**i***, *defghi**j***, *defghij**k***, *efghijk*, *fghijk*, *fghijk**l***, *ghijkl*, *hijkl*, *ijkl*, *jkl*, *kl*, *l*.

The list of vertices in the order in which they are first visited is:



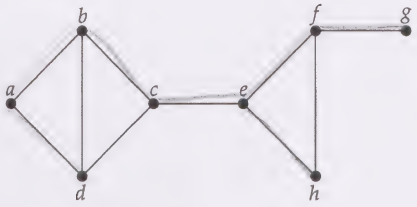
■

Breadth-first search on a connected graph

As for a depth-first search, the algorithm for manipulating the queue in a breadth-first search — with 'starting vertex' replacing 'root vertex' in Step 1 — can also be applied to any connected graph.

Example 3.11

Consider the following graph.

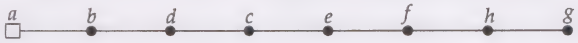


In this case, starting from *a*, the queue could evolve as follows:

a, ***ab***, *ab**d***, *bd*, *bd**c***, *dc*, *c*, *c**e***, *e*, *e**f***, *ef**h***, *fh*, *fh**g***, *hg*, *g*.

Other searches are possible.

The vertices are visited according to the following list:

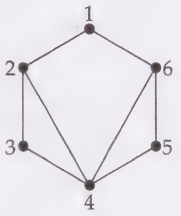


If, at each stage, we also consider the edge that joins the first vertex of the queue to the new vertex being added at the end of the queue, then we obtain the following spanning tree:

Such a tree is called a **breadth-first search spanning tree**. ■

Problem 3.8

Draw a depth-first search spanning tree and a breadth-first search spanning tree for the graph in the margin, starting at vertex 1.



Time complexity functions

The following result can be proved without too much difficulty, but we do not have space to do so here.

Theorem 3.2

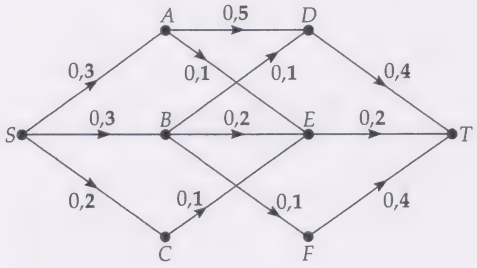
The time complexity functions for depth-first search and breadth-first search of a connected graph with m edges are both of order $O(m)$.

Depth-first search versus breadth-first search

For many problems involving graphs it is important that we choose the correct searching procedure, since the wrong decision can result in a great deal of unnecessary work and expense. Unfortunately, no *general* rule can be given as to which procedure to choose, since each method has its advantages and disadvantages, depending on the problem in hand. We illustrate this by means of two algorithms you have met before.

Labelling procedure

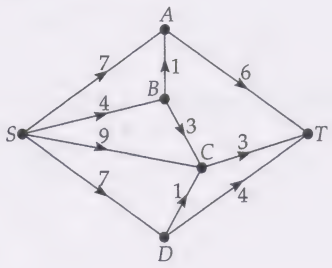
In *Networks 1* we described the *labelling procedure* — a systematic step-by-step method for building up a maximum flow in a given basic network by finding a succession of flow-augmenting paths. For example, in the following network, we can increase the flow by 3 along *SADT*, by 1 along *SBDT*, by 2 along *SBET*, and by 1 along *SCEBFT*, giving a maximum flow of 7.



In order to find each flow-augmenting path, we used what amounts to a *depth-first search*. For example, we find the flow-augmenting path *SADT* by increasing the flow from *S* to *A*, calculating how much of this increase can be transmitted to *D*, and then calculating how much can be transmitted to *T*. If at any stage no increase of flow is possible, then we backtrack and try another vertex. At no stage in finding this path are we concerned with the flows from *S* to *B* or to *C*, so a breadth-first search would be very wasteful of effort, and inappropriate for this problem. For large networks, we can often make significant savings in effort and expense by using depth-first search rather than breadth-first search.

Shortest path algorithm

In *Networks 2* we discussed the *shortest path algorithm* — a method for finding the shortest path between two given vertices of a weighted graph or digraph. For example, in the following network, the shortest path from *S* to *T* is *SBCT*, with total length 10.



Since the distance from *S* to each vertex depends on the distances to the previous vertices, we first need to determine the distance from *S* to each of its neighbours, before going on to *their* neighbours, and so on. Thus the shortest path algorithm amounts to a *breadth-first search* on the graph or digraph. A depth-first search would involve taking a path directly to *T*, ignoring the other vertices closer to *S* on the way, and would be inappropriate for this problem.

After studying this section, you should be able to:

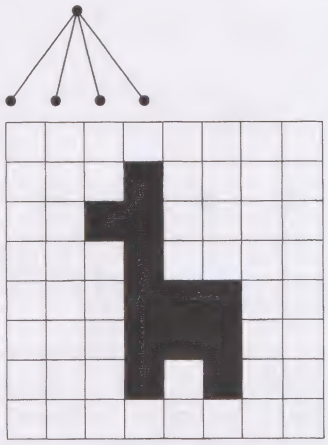
- understand what are meant by *binary tree, left subtree, right subtree, height of a binary tree, balanced binary tree* and *binary tree data type*;
- grow a binary tree from a list;
- understand what is meant by a *binary search tree* and appreciate its advantages as a store for searching;
- perform a *depth-first search* and a *breadth-first search* on any connected graph;
- compute time complexity functions, and their orders, for certain algorithms.

4 Quad trees

In this section we are concerned with the representation of images on a computer screen, and with the operations (rotation, reflection, etc.) that can be carried out on such images. The representations can be modelled by means of trees, called *quad trees*, and we can interpret each operation on the image as an operation on the associated quad tree.

4.1 Images on a computer screen

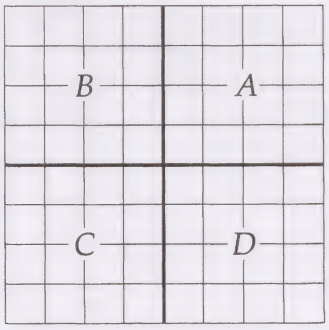
Binary trees are not the only type of tree used to store and manipulate information. In computer graphics a **quad tree**, with *four* branches descending from each vertex (other than the end-vertices), is used to hold information about an image that can be displayed on a two-dimensional computer screen. The screen is modelled by a rectangular grid consisting of a large number of small squares, called **pixels**; in the case of a monochrome screen, each pixel can be either filled in (a black pixel) or left empty (a white pixel). A typical computer screen displays images on a grid of many thousands of pixels, and so a large amount of information must be stored for each image. In order to be able to draw diagrams, we shall work with much smaller 'screens'; for example, the image of an animal (a giraffe, maybe) is shown on a grid of $8 \times 8 = 64$ pixels.



Quad trees were first used to store screen images in the 1970s.

The information for this image can be held in computer memory in a stack or list. However a more efficient way of storing screen images is to use a quad tree. For an 8×8 screen, the quad tree is constructed as follows.

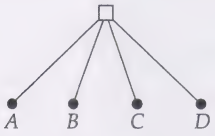
We first divide the 8×8 grid into four *quadrants* A, B, C and D, each consisting of 4×4 pixels:



In any grid subdivided into four quadrants, quadrants A and B are the *upper* quadrants and quadrants C and D are the *lower* quadrants.

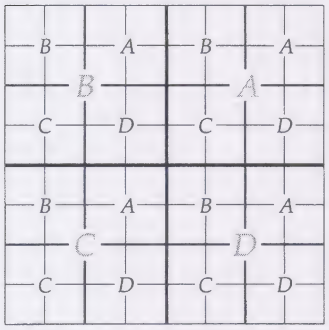
We label the quadrants in anticlockwise order, with A denoting the top-right quadrant, B the top-left quadrant, C the bottom-left one and D the bottom-right one. This situation can be represented by a rooted tree, with

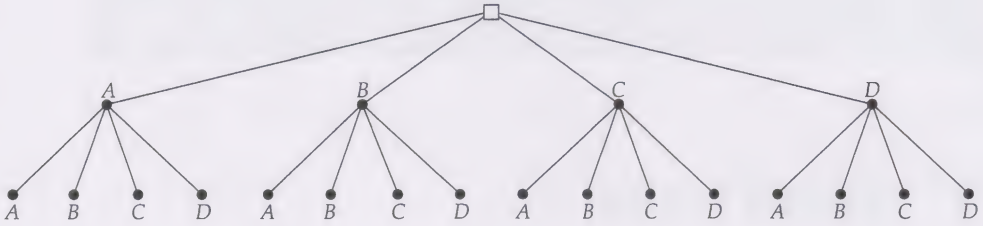
the end-vertices corresponding to the quadrants in anticlockwise order *A*, *B*, *C*, *D*.



Note that *A* is the *label* of the leftmost vertex, and *not* what is stored in it. The left-to-right sequence of labels *A*, *B*, *C*, *D* distinguishes the four vertices in the same way as the labels *left* and *right* distinguish the pairs of vertices in a binary tree.

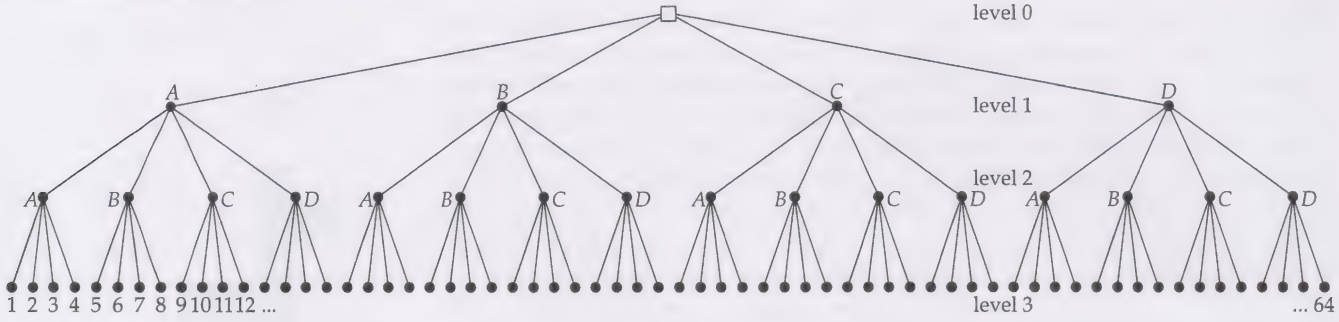
We now repeat the above process, subdividing each 4×4 quadrant into four 2×2 quadrants, labelled *A*, *B*, *C* and *D* as before. In the graph representation, we adjoin four subtrees, each identical to the one above, to each end-vertex of the above tree.





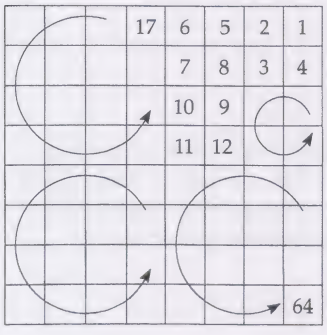
This process is recursive. We continue subdividing until the resulting quadrants consist of a single pixel. Each subdivision adds another level to the quad tree.

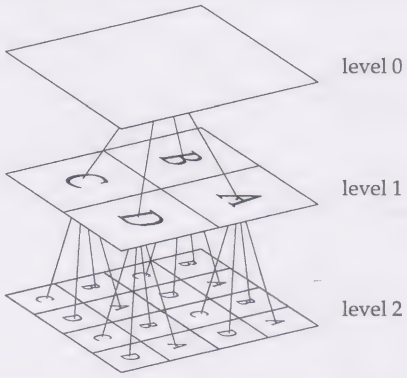
Returning to our 8×8 example, we can make just one further subdivision. Each 2×2 quadrant can be subdivided into four 1×1 quadrants, and so in the graph representation we adjoin 16 subtrees. Hence the quad tree for an 8×8 screen is as follows:



The above diagram has 64 end-vertices, all on level 3 of the quad tree. These are labelled 1, 2, 3, ..., 64. The vertex labelled 1 corresponds to the cell labelled 1, and so on, as shown on the left below. On the right below is a three-dimensional picture of levels 0, 1 and 2 of the quad tree. It shows how the vertices at each level relate to the successive subdivisions of the screen.

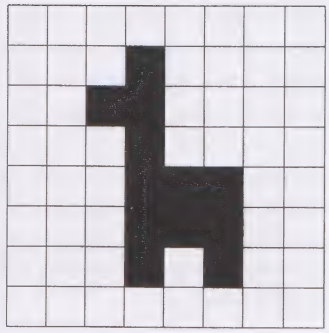
The path from the root vertex to an end-vertex can be represented by a string of letters corresponding to the labels of the non-root vertices in the path. For example, the path to the vertex corresponding to the top right pixel in our 8×8 example is represented by *AAA*.

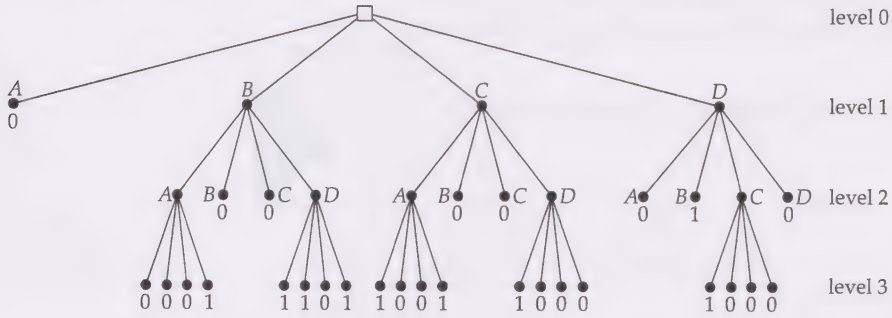




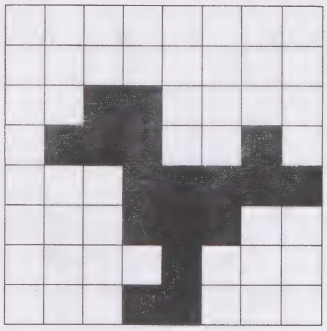
We can store information about a particular image on a screen by writing 1 (for black) or 0 (for white) at each end-vertex of the quad tree, indicating the colour of the corresponding pixel on the screen.

Most images can be stored without using a *full* quad tree. This is because often there are quadrants composed entirely of pixels of a single colour. For instance, there may be 4 × 4 or 2 × 2 quadrants of white pixels and we indicate this on the quad tree by writing 0 at a single vertex at the correct level in the quad tree; no edges leading to subtrees for that quadrant are then needed. For example, the giraffe on the 8 × 8 screen is stored by the following quad tree of height 4.





We can think of the full quad tree as having been 'pruned'. Whenever the four vertices in a set are all assigned the value 1 or all assigned the value 0, the four vertices and their incident edges can be removed and the tree pruned back to the single parent vertex with 1 or 0, as appropriate, written next to it.



Problem 4.1

Draw the pruned quad tree used to store the screen image of a robin shown.

Problem 4.2

Which images need a full (i.e. unpruned) quad tree to describe them?

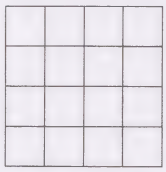
Definitions

A ***k*-screen** consists of $2^{k-1} \times 2^{k-1}$ pixels.

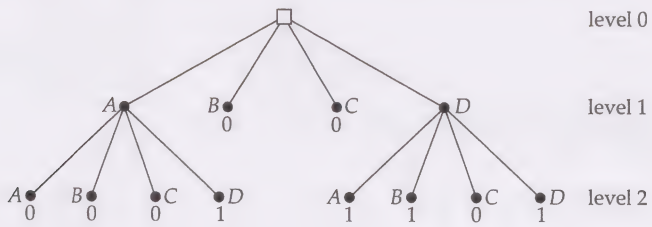
A **quad tree** that represents a *k*-screen is a rooted tree with at most *k* levels. Each end-vertex has 1 or 0 stored at it, denoting that the pixel (or quadrant of pixels) it represents is black or white, respectively. Each other vertex has a set of four vertices below it, ordered from left to right, this ordering being denoted by the labels *A*, *B*, *C*, *D*. At each level, each set of four vertices corresponds to four quadrants of the screen — the *A*-vertex to the top-right quadrant, the *B*-vertex to the top-left quadrant, and so on, in an anticlockwise direction.

The columns of a K-screen are numbered from left to right and the rows from top to bottom.

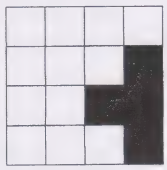
In order to reproduce the image stored in a quad tree on a screen, we use a *depth-first* search on the quad tree. For example, for the above pruned quad tree representing the giraffe image, the search first reaches the *A*-vertex at level 1 of the tree. This is an end-vertex with value 0, indicating that the top-right 4 × 4 quadrant consists only of white pixels. The next vertex reached, the *B*-vertex at level 1, leads to the information for the top-left 4 × 4 quadrant. This vertex is the root of a quad subtree of height 3, shown below.



the 4 × 4 *A*-quadrant



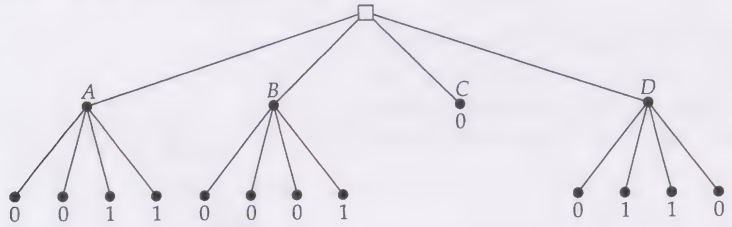
Depth-first search of this subtree begins at the A-vertex at level 1, leading next to the A, B, C, D vertices at level 2, from which the image in the top-right 2×2 quadrant of the top-left 4×4 quadrant can be obtained as shown in the margin. Next the search reaches B and C at level 1, and the 0s here indicate that the top-left and bottom-left 2×2 quadrants of the top-left 4×4 quadrant are both all white. Then the search reaches D at level 1, leading to the A, B, C, D vertices at level 2, from which we obtain the image in the bottom-right 2×2 quadrant of the top-left 4×4 quadrant as shown in the margin. Hence the complete top-left 4×4 quadrant is as shown.



The entire 8×8 screen image of a giraffe is retrieved by completing the depth-first search.

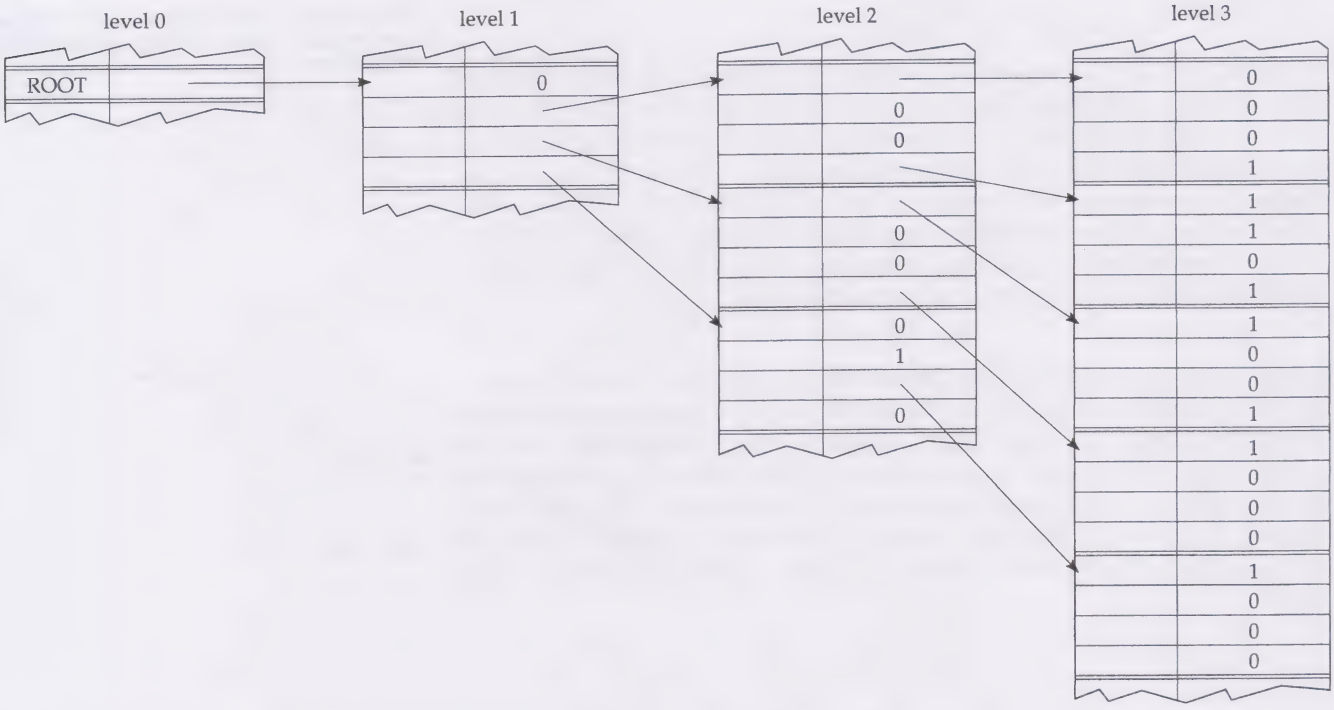
Problem 4.3

The following quad tree is used to store an image on a 4×4 screen. Reproduce the screen, showing the image.



We conclude this introduction to quad trees by taking a brief look at how we store a quad tree in computer memory. We represent each set of four vertices A, B, C, D by a stack of four cells. The A, B, C, D order is preserved by having A at the top of the stack, and so on, down to D. Each cell corresponding to an end-vertex contains either 1 or 0, and each of the other cells contains the address of the top cell of the stack representing the set of four vertices below it. For example, the 8×8 giraffe image is stored on tape as follows:

With the various sections of the tape pictured in this way, you can see the quad tree structure — laid on its side as it were.



A depth-first search algorithm on a store arranged in this way is based on one simple task:

> move down a stack of four cells, starting at an A-vertex at the top of the stack.

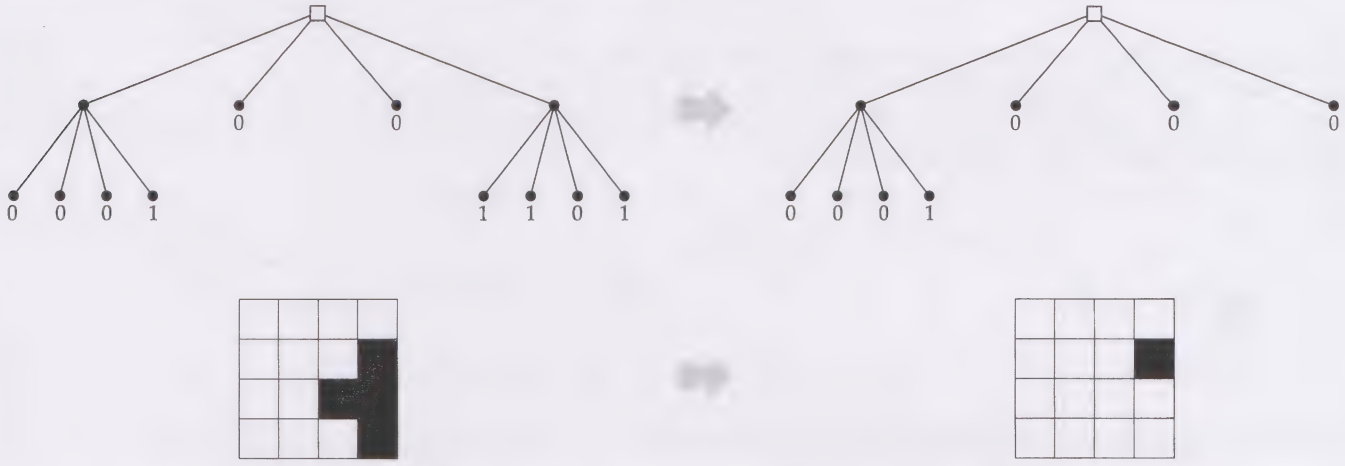
When 1 or 0 is found, we fill in the corresponding appropriately sized quadrant with the corresponding single colour. When a forwarding address is found, we move to the next stack of four cells and complete the same task again. Each time we move to the next level, we need to push a return address onto a temporary stack so that, when we find no more forwarding addresses, we can use the return address to backtrack to the correct previous stack of four cells and continue to move down it.

A quad tree generally requires fewer storage cells than other forms of data store, and quad trees are easily searched and manipulated. The algorithms to store and display images using quad trees are recursive and relatively straightforward. Recently a lot of work has gone into refining such algorithms and producing machines designed specifically to work with quad trees. Quad-tree systems exist that can store and display, in real time, sequences of images on large screens.

4.2 Manipulating images

One simple way to change an image using its quad tree is to blank out (make white) a certain quadrant. All we need to do is to delete the subtree at the relevant vertex and write 0 in its place, as follows:

Similarly, we can make a quadrant black by deleting the appropriate subtree and writing 1 in its place.

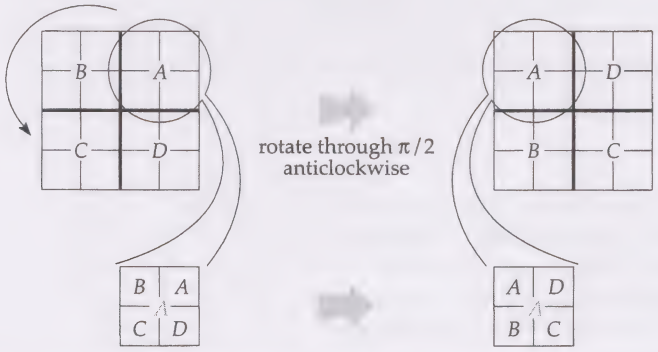


Indeed, a number of standard manipulations of graphic images can easily be performed on a quad tree. Quad trees have an inbuilt sense of rotation (the anticlockwise order for the quadrants) and of magnification (since moving from one level to the next changes the length of the sides of each quadrant by a factor of 2). So the rotation and magnification of graphical images correspond to straightforward manipulations of the vertices and levels of a quad tree.

Rotating an image

Consider a 3-screen and the principal quadrants in a subdivision of the screen.

The *principal quadrants* of a *k*-screen are those that correspond to the vertices at level 1 of its full quad tree.



To rotate any image anticlockwise through a right angle, we rotate it about its centre. Thus, in the quadrants of *each* subdivision, the

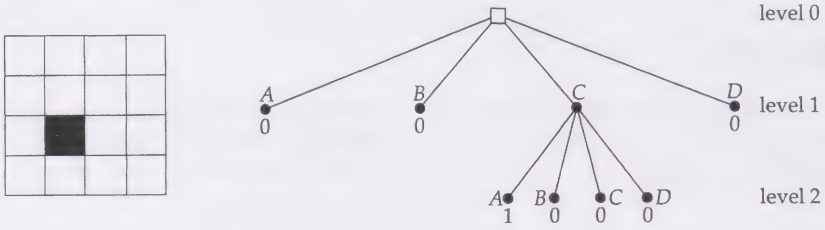
A-quadrant is rotated into the place previously occupied by the B-quadrant, the B-quadrant is rotated into the C-quadrant, the C-quadrant is rotated into the D-quadrant, and the D-quadrant is rotated into the A-quadrant. We can represent these rotations by a cycle:

$$A \to B \to C \to D \to A$$

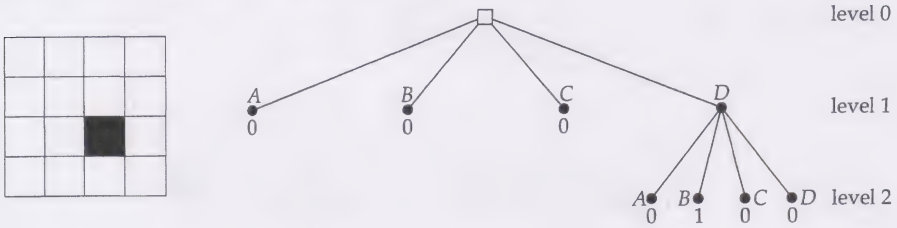
In the quad tree, such a rotation corresponds to a similar cycling of whatever is stored at the vertices in each of the sets of four. Thus, whatever was attached to an A-vertex is now attached to the B-vertex, and so on, until whatever was attached to the D-vertex is now attached to the A-vertex. This is carried out for each set of A, B, C, D vertices in the quad tree.

Example 4.1

Consider the following image on a 3-screen and its corresponding quad tree:



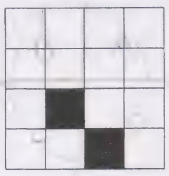
To rotate this image anticlockwise through a right angle, we cycle ($A \to B \to C \to D \to A$) whatever is attached to the four vertices at level 1 of the tree, and then repeat the cycling on the set of four vertices at level 2. We thus obtain the following image and quad tree:



Note that we keep the left-to-right ordering for the labels A, B, C, D. ■

Problem 4.4

Given the following image on a screen, construct its quad tree and the quad tree for a *clockwise* rotation of the image through a right angle.

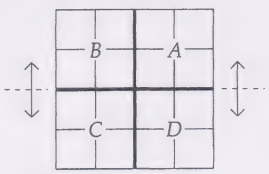


Problem 4.5

How do we manipulate a quad tree to produce a rotation of a given image through two right angles?

Reflecting an image

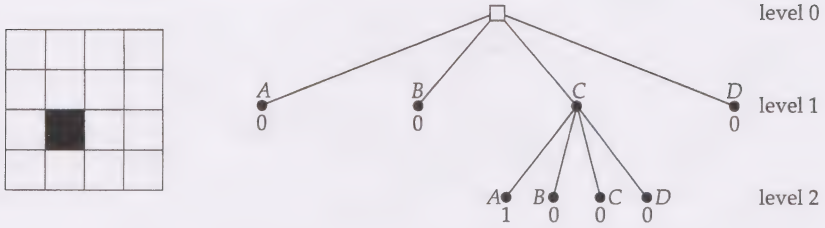
To reflect an image in a horizontal or vertical line in the plane, we interchange the quadrants in an appropriate way. For example, consider a reflection about a horizontal line through the middle of a 3-screen. In order to achieve this, we interchange the quadrants A and D and the quadrants B and C. In the quad tree, at each level we interchange whatever is at the vertices, according to the scheme



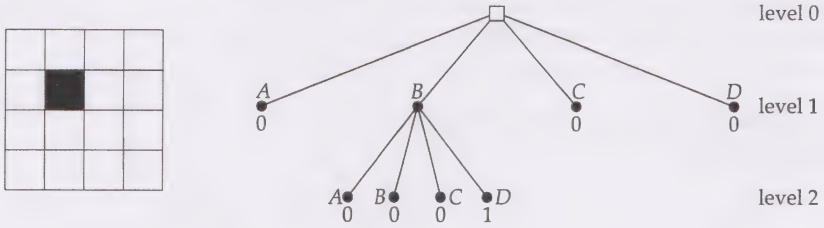
$$A \leftrightarrow D \quad \text{and} \quad B \leftrightarrow C.$$

Example 4.2

Consider the following image and its quad tree:

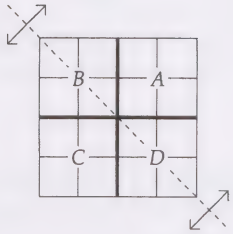


If we reflect about the horizontal line through the middle of the screen, we get the following screen and quad tree:



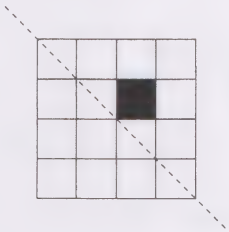
A similar process holds for a reflection about a diagonal. In the case shown, the quadrants B and D remain unchanged, while the quadrants A and C are interchanged. The scheme this time is just

$$A \leftrightarrow C.$$



Problem 4.6

Construct the quad tree corresponding to the following image, and the quad tree corresponding to its reflection about the top-left to bottom-right diagonal shown.

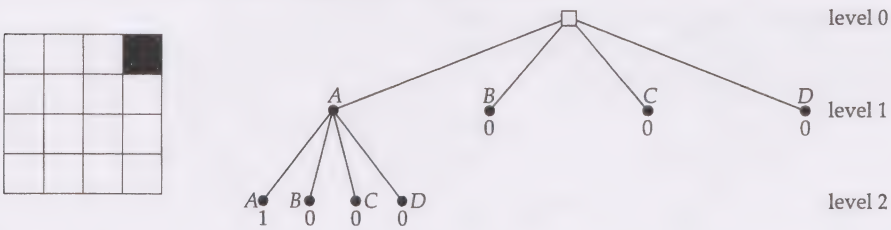


Magnifying an image

Moving up or down a level in a quad tree changes the length of the sides of the appropriate quadrants by a factor of 2. Thus, to magnify the image in one of the principal quadrants by a factor of 2, we can interpret the subtree for that quadrant as the quad tree for the whole screen.

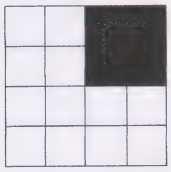
Example 4.3

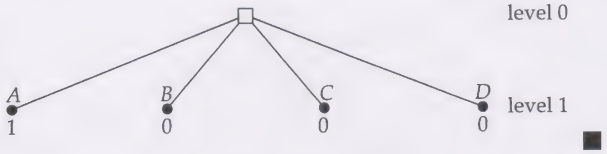
Consider the following image and its quad tree:



In order to magnify the image in the principal A-quadrant by a factor of 2, we detach the subtree from the A-vertex at level 1 and consider it as a quad tree for the whole screen. The top-right black pixel is then

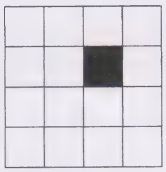
magnified by a factor of 2, in both depth and width, and becomes a black 2×2 quadrant. And the three white pixels in the quadrant are also magnified by a factor of 2, and become white 2×2 quadrants.





Problem 4.7

Construct the quad tree corresponding to the image shown, and the quad tree corresponding to magnifying the principal A-quadrant to fill the whole screen. Draw the screen image corresponding to the magnification.



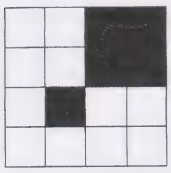
Magnification by other powers of 2 is also possible by moving further down the levels of the quad tree and detaching the subtrees.

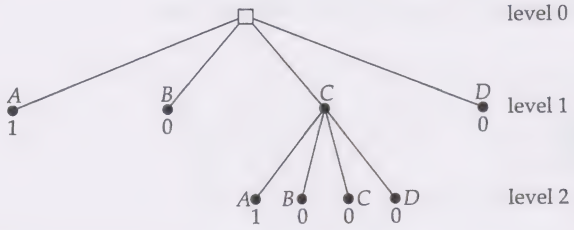
Reducing an image

In order to reduce an image by factors of 2, we move all vertices down one or more levels. To see what this means, consider the following example.

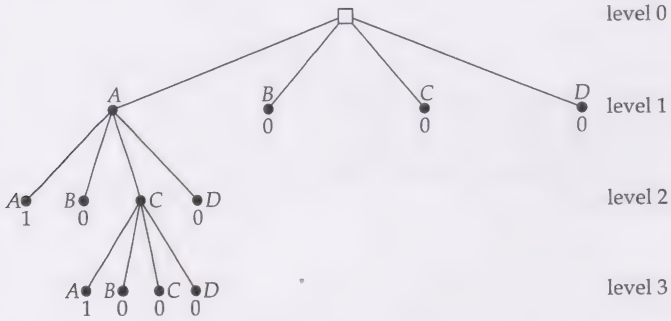
Example 4.4

Consider the following image on a 3-screen, together with its quad tree, and suppose that we want to reduce the image so that it all fits in the principal A-quadrant.

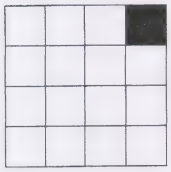


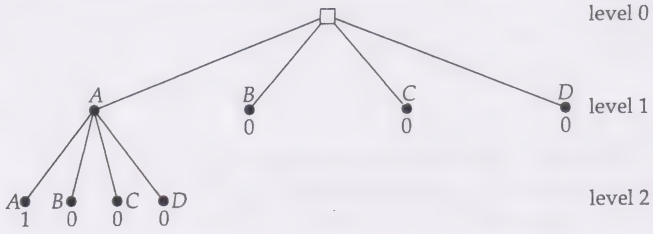


To do this, we construct a new tree that has the above tree attached to its level 1 A-vertex, the other three vertices at level 1 all having the value 0:



This quad tree has four levels, which is one too many for a 3-screen. We therefore eliminate all subtrees with roots at level 2, writing 0 in their place, to obtain the following quad tree and corresponding image:



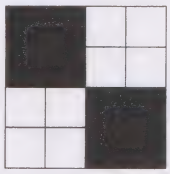


Note that the original 2×2 quadrant of black pixels has been reduced to a single pixel, whereas the single pixel in quadrant C has vanished altogether; it is impossible to reduce a single pixel, and hence it is discarded. ■

In fact, the image in any 2×2 quadrant that is not made up of four black pixels is lost when we reduce the whole image by a factor of 2.

Problem 4.8

Construct the quad tree for the 3-screen image shown, and use it to construct a tree for the image reduced by a factor of 2 into the principal B-quadrant. Draw the screen image corresponding to the new quad tree.



Reduction by other powers of 2 is also possible by applying the above technique repeatedly.

Approximating an image

The final part of the above reduction technique — removal of the bottom level from a quad tree — is often used on its own to the following effect. If one or more levels are removed from the bottom of a quad tree, then the image loses resolution and becomes approximate, but becomes simpler to manipulate and analyse because of the reduction in the data structure. This is useful in pattern recognition, where we want fast analysis but are not concerned with fine detail.

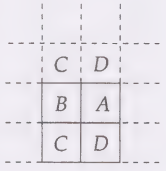
4.3 Analysing images

In analysing images, we are often interested in the colours of the pixels that surround a given one. For instance, if a black pixel is surrounded entirely by white pixels, then we may wish to make it white as well, considering it to be a blemish and not part of the actual image. The *north neighbour algorithm* introduced in this subsection is part of the algorithm needed to do this.

Finding a north neighbour

Starting at a given pixel, how do we find the pixel immediately above it, if there is one? We call such a pixel the **north neighbour**.

Consider the diagram of a 2×2 pixel quadrant in the margin. The north neighbour of pixel C is pixel B in the same 2×2 quadrant. The north neighbour of pixel B (unless it lies on the top edge of the screen) is pixel C from *another* 2×2 quadrant. We say that the pixels B and C are **opposites** of each other, as far as being north neighbours is concerned; a B-pixel is always north of a C-pixel, and a C-pixel is always north of a B-pixel. Similarly, the pixels D and A are opposites of each other.

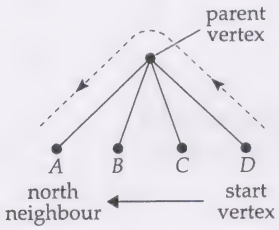


These ideas also apply to larger blocks of pixels. For blocks of 2×2 pixels, the B-quadrant and C-quadrant are opposites, and the A-quadrant and D-quadrant are opposites, since they are north neighbour quadrants of each other.

In a quad tree, the search for a north neighbour pixel starts at a given end-vertex at the bottom of the tree and moves along the path to the end-vertex that represents its north neighbour.

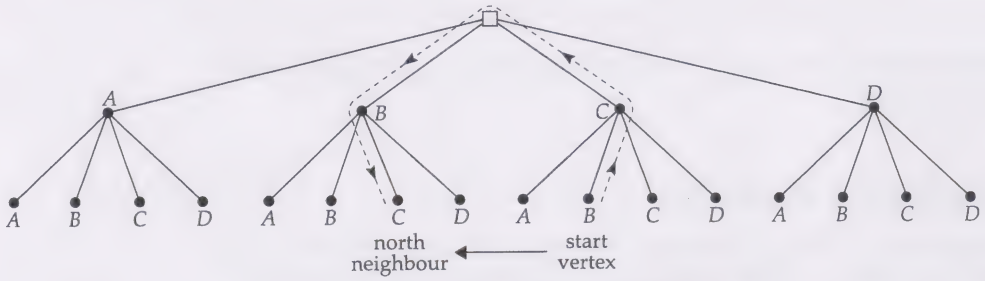
We consider only full quad trees, so that all the end-vertices are at the bottom level of the tree.

If we start at a lower vertex (*C* or *D*), we simply move across to the *opposite* upper vertex (*B* or *A*, respectively) in the set of four vertices. We can think of this as a path up to the parent vertex and then down to the opposite vertex.



If we start at an upper vertex (*A* or *B*), then we move up the tree until we reach a point from which we can move down again to the opposite lower vertex (*D* or *C*, respectively) in the appropriate different set of four vertices.

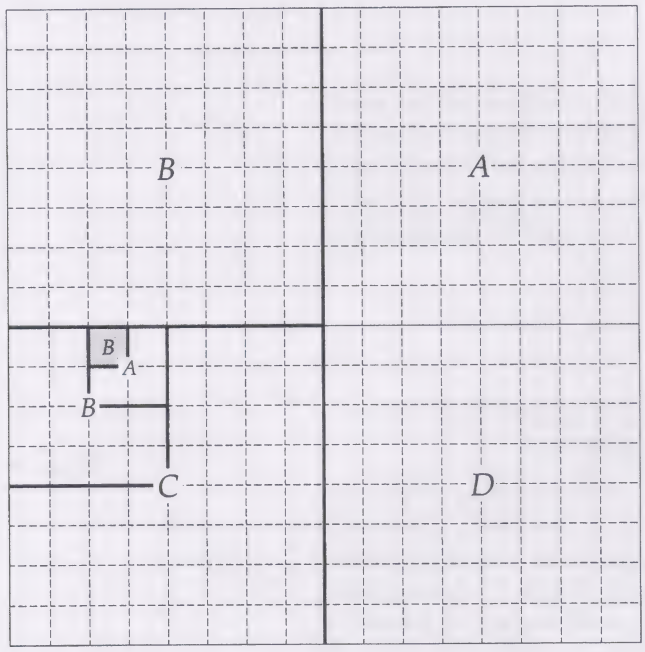
B	A	B	A
C	D	C	D
B	A	B	A
C	D	C	D



finding the north neighbour of the shaded *B*-pixel

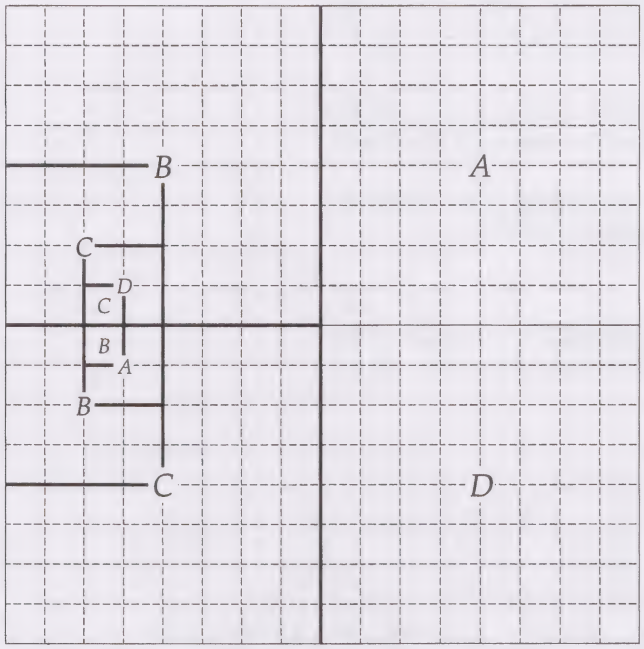
Example 4.5

Consider finding the north neighbour of the shaded *B*-pixel in the following diagram.

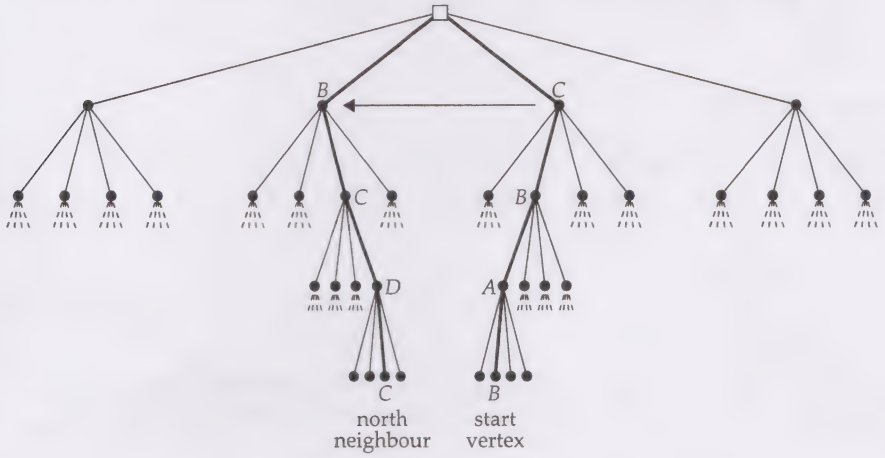


This *B*-pixel appears at the top of a 2 × 2 block. The 2 × 2 block is the *A*-quadrant of a 4 × 4 block, and hence appears at the top of this block. The 4 × 4 block is a *B*-quadrant of an 8 × 8 block, and hence appears at the top of the block. The 8 × 8 block is a *C*-quadrant of a 16 × 16 block, and hence has its north neighbour 8 × 8 quadrant in that 16 × 16 block. This is the *B*-quadrant (the opposite of *C*) that lies directly to the north of it. We now have to move down through subdivisions of this 8 × 8 *B*-quadrant until we reach the north neighbour pixel we need.

We can do this by taking, at each step, the quadrant *opposite* to the one that we met on the way up. From the 8×8 B-quadrant, we move down to its 4×4 C-quadrant. Note that this is the north neighbour quadrant of the 4×4 B-quadrant through which we came up. We then move to the 2×2 D-quadrant, again the north neighbour quadrant of the 2×2 A-quadrant. Finally we move to the C-pixel which is the required north neighbour pixel.



In terms of the quad tree, we trace out the following path:



Starting at a B-pixel, we move up the tree, noting the vertices as we pass them, until we reach a C-vertex or D-vertex (in this case, a C-vertex). We then move across to the opposite vertex in the set of four, in this case the B-vertex, by moving up to the parent vertex and then down. Finally, we move down through vertices that are the opposites, at each level, of the ones that we met on the way up. ■

If, as we move up the quad tree, we reach the root before reaching a C-vertex or D-vertex, then the pixel at which we started has no north neighbour, as it lies on the top edge of the screen.

The simplest way that we can keep track of the vertices we meet on the path up the quad tree is to push them onto a stack as we meet them. They are then in the correct order for finding the opposite vertices for the path down the quad tree.

We now present the above ideas in the form of an algorithm.

North neighbour algorithm: for finding the north neighbour of a vertex of a quad tree

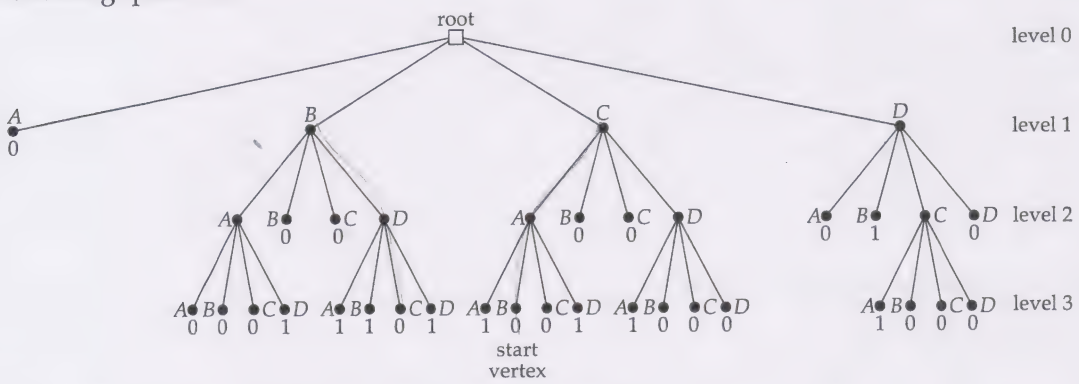
STEP 1 Set up an empty 'path' stack to keep track of the vertices that we pass. Make the start vertex the 'current' vertex.

STEP 2 If the 'current' vertex is the root vertex, and a lower (*C* or *D*) vertex is not on the TOP of the 'path' stack, then no north neighbour exists, so STOP.
Otherwise, PUSH the 'current' vertex label onto the 'path' stack. Move up to the parent vertex and make it the 'current' vertex. If the TOP of the 'path' stack is not a lower (*C* or *D*) vertex, repeat Step 2.

STEP 3 POP a vertex label from the 'path' stack. Move down to the vertex with the *opposite* label to the one just taken off the stack, and make it the 'current' vertex.
Repeat Step 3 until the 'path' stack is empty. The 'current' vertex is then the north neighbour of the start vertex.

Example 4.6

We shall use the north neigbour algorithm to determine the colour of the north neighbour of the pixel represented by the indicated vertex of the following quad tree.

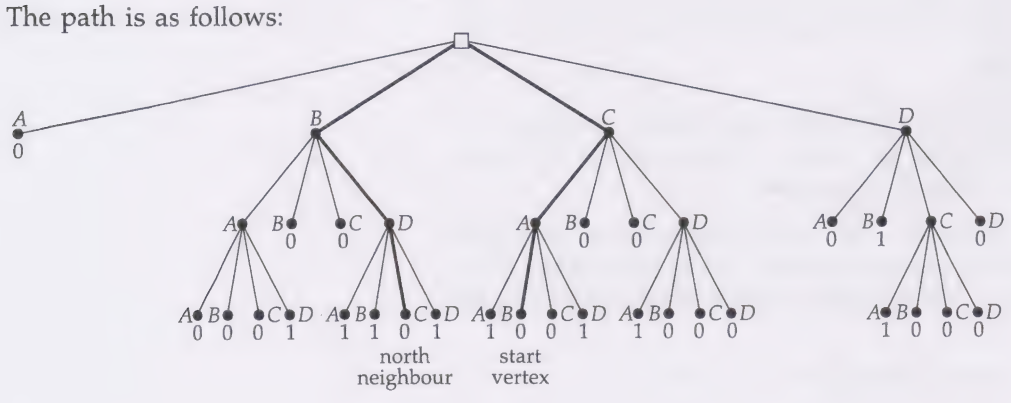


The following table traces the steps in the algorithm:

	step	current vertex label	level	path stack	top of stack
	1	*B*	3	empty	–
moving up the tree	2	*A*	2	*B*	*B*
	2	*C*	1	*BA*	*A*
	2	root	0	*BAC*	*C*
moving down the tree	3	*B* (opposite to *C*)	1	*BA*	*A*
	3	*D* (opposite to *A*)	2	*B*	*B*
	3	*C* (opposite to *B*)	3	empty	–

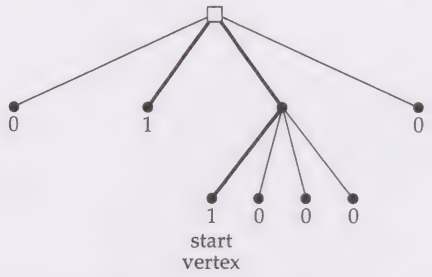
A lower vertex is now on top of the stack, so we move to Step 3.

The path is as follows:



The north neighbour pixel is therefore white. ■

In the above example, the quad tree was 'pruned', and so we were lucky to be able to work down the tree to the exact north neighbour pixel. However, we are not often interested in finding the exact north neighbour pixel, but rather what its colour is. To do this, we *can* use a pruned quad tree since, as we work down the tree, if we come to an end-vertex before the path stack is empty, it tells us the colour of all the pixels in a quadrant that contains the north neighbour pixel, and hence we know its colour. For example, consider the pruned quad tree below. If we start at the vertex indicated, then the algorithm traces out the path shown, which stops at level 1. However, the 1 at the end-vertex reached tells us that the north neighbour pixel must be black. Thus the north neighbour algorithm is easily adapted to find just the *colour* of the north neighbour pixel.

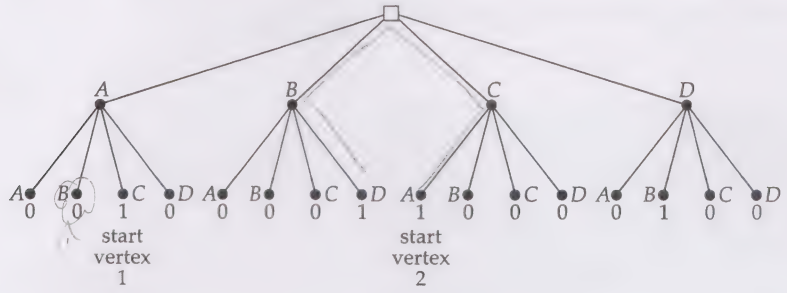


The time complexity function of the north neighbour algorithm is straightforward to calculate. As we work up the quad tree, we compare each vertex label with A, B, C and D, in order to know what to do next, so at worst we make four comparisons at each vertex. For a k-screen, the quad tree has k levels and, at worst, we work up through $k - 1$ vertices; this gives a maximum of $4(k - 1)$ comparisons in total. Therefore the time complexity function for the north neighbour algorithm applied to a k-screen is $T(k) = 4(k - 1)$, of order $O(k)$.

No comparisons are made on the way back down the quad tree.

Problem 4.9

The following quad tree represents an image on a 3-screen. For each of the two pixels corresponding to the start vertices indicated, use the north neighbour algorithm to determine the colour its north neighbour pixel.



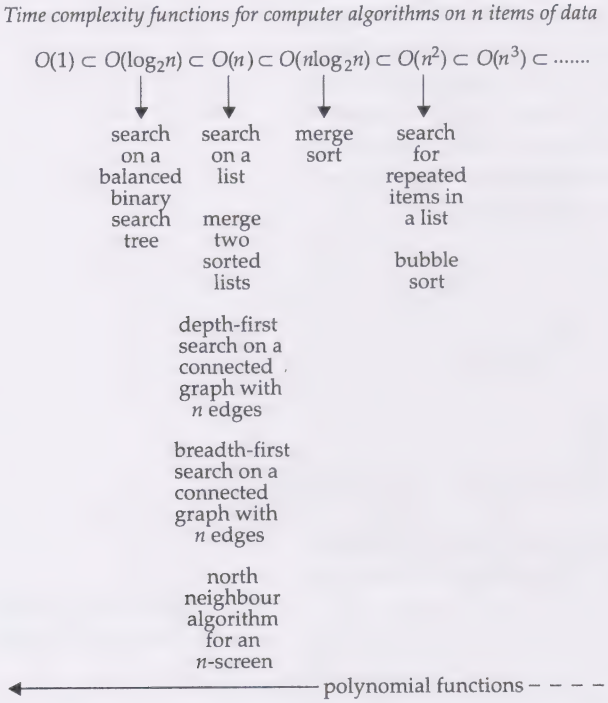
Finding other neighbours

The north neighbour algorithm can be adapted to yield (the colours of) the other seven neighbouring pixels — the south, east, west, north-east, north-west, south-east and south-west neighbour pixels. For example, to change it to an algorithm that finds the south neighbour, we simply change the roles of the upper (A and B) and lower (C and D) vertices in Step 2.

The time complexity function for each of the eight neighbour algorithms applied to a k-screen is $T(k) = 4(k - 1)$, and so finding all eight neighbours has time complexity function $T(k) = 8 \times 4(k - 1) = 32(k - 1)$. Thus the algorithms for finding one, or all, of the eight neighbours of a given pixel are all of order $O(k)$.

4.4 Time complexity functions for computer algorithms

We have now completed our examination of computer algorithms for manipulating and searching data. We can summarize our discussions by comparing their efficiencies, by seeing where their time complexity functions fit in the order hierarchy.



All of the algorithms are efficient (fast) in the sense that their time complexity functions are contained in sets whose defining function is polynomial. In the language of the *Introduction* unit, they are all polynomial-time algorithms.

4.5 Computer activities

The computer activities for this section are described in the *Computer Activities Booklet*.

After studying this section, you should be able to:

- explain what are meant by a *k-screen* and a *quad tree*;
- understand how a quad tree is used to store a screen image and how a quad tree can be represented in a computer store;
- explain how to rotate, reflect, magnify, reduce and approximate an image by adapting its quad tree;
- apply the *north neighbour algorithm* to find the north neighbour of a given pixel.

5 Branch-and-bound methods

In computer science, trees are used as structures for storing data, and we have looked at a number of ways of searching such trees. Trees are also useful as structures for describing methods for solving problems in discrete mathematics, where the method of solution involves searching such a tree in a systematic way.

5.1 State space trees

One method we have already met is that of *divide and conquer*, where we break down a problem into subproblems, solve these subproblems, and then combine the solutions to the subproblems into a solution for the original problem. The method is recursive, in that the subproblems are themselves solved using the divide-and-conquer technique. This method of solution can be described by a tree, as the following example illustrates.

Example 5.1

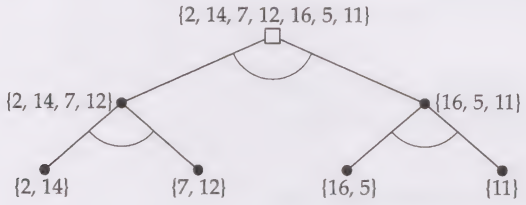
We shall use the divide-and-conquer method to find the largest number in the set

{2, 14, 7, 12, 16, 5, 11}.

The divide-and-conquer method splits the set into two subsets,

{2, 14, 7, 12} and {16, 5, 11}.

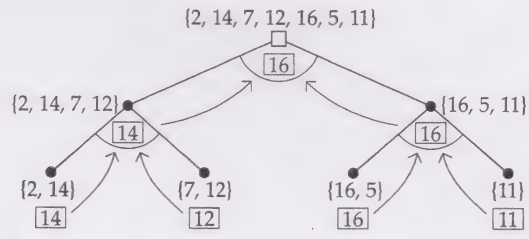
thereby giving us two subproblems. These subproblems are themselves solved by splitting the subsets into two, giving us the hierarchy of problems illustrated in the following *state space tree*:



The linking curves in the diagram indicate that the subproblems at both the left and right vertices immediately below a curve must be solved in order to solve the problem at the vertex immediately above that curve.

The subproblems at the end-vertices of the state space tree are easily solved. For the other vertices of the tree, the problem of finding the largest number in the set is easily solved provided that we have solved the problems for the vertices below it — that is, for all of its *children* vertices.

We solve this problem by conducting a depth-first search of the state space tree. We start at the root. There is no immediate solution for the initial problem, so we go down left. There is no immediate solution for the set {2, 14, 7, 12}, so we go down left. At the end-vertex we easily find the solution 14 (the larger of 2 and 14). We now backtrack. There is still no solution for {2, 14, 7, 12}, so we go down right to the end-vertex with solution 12 (the larger of 7 and 12). We backtrack to the set {2, 14, 7, 12}, which now has solution 14 (the larger of 14 and 12). We now backtrack to the root. There is still no solution, so we go down right, and then down left, to the {16, 5} end-vertex. This set has solution 16. We backtrack and go down right to the set {11} with solution 11. We backtrack and now the set {16, 5, 11} has solution 16 (the larger of 16 and 11). We backtrack, and the initial problem has solution 16 (the larger of 14 and 16). The solution process is illustrated below:



■

Many problems in computer science and in graph and network theory can be solved with the help of searches of state space trees. These trees can have two types of vertex. They can have *AND vertices*, as in Example 5.1, for which the (sub)problem corresponding to a parent vertex can *only* be solved once *all* the subproblems corresponding to its children vertices have been solved. AND vertices are drawn with curves linking the branches joining

parent vertex to its children vertices. State space trees can also have *OR vertices*, for which the (sub)problem corresponding to a parent vertex can be solved when *any one* of the subproblems corresponding to its children vertices has been solved. OR vertices have no linking curves below them.

Definition

A **state space tree** for a problem is a rooted tree in which the root vertex corresponds to the problem as a whole and the other vertices correspond to subproblems. The subproblems at the end-vertices are ones that can be solved immediately. Each parent vertex is:

either an **AND vertex**, in which case its (sub)problem can be solved immediately once *all* the subproblems at its children vertices have been solved;

or an **OR vertex**, in which case its (sub)problem can be solved immediately once *any one* of the subproblems at its children vertices has been solved.

An **AND tree** is a state space tree all of whose parent vertices are AND vertices.

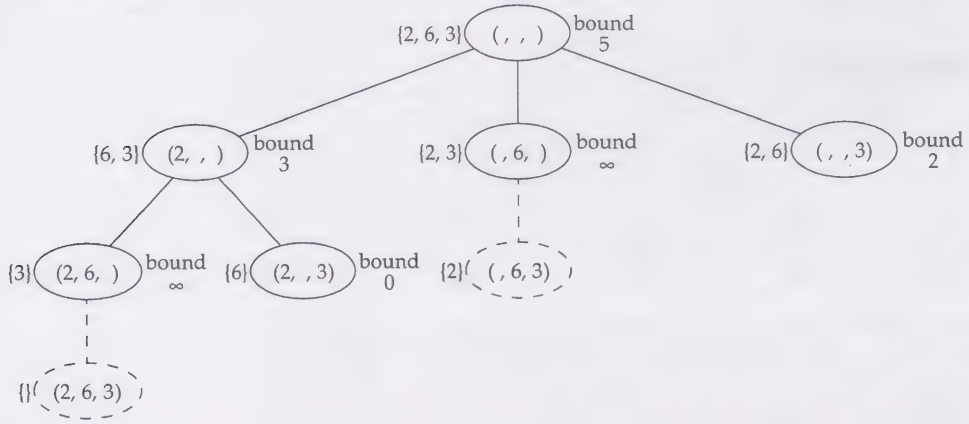
An **OR tree** is a state space tree all of whose parent vertices are OR vertices.

And **AND/OR tree** is a state space tree that has a mixture of parent vertex types.

The *divide-and-conquer method* corresponds to an AND state space tree, since *every* subproblem has to be solved in order to solve the initial problem. In other words, a search of an AND state space tree has to be *exhaustive* — every vertex must be visited. The *branch-and-bound method*, however, corresponds to an OR state space tree, a search of which need not be exhaustive.

To explain how the branch-and-bound method works, consider the following example.

Example 5.2



This is a state space tree for the problem of choosing numbers from the set {2, 6, 3} that add up to exactly 5.

The original problem is shown at the root vertex. The {2, 6, 3} term on the left of the vertex represents the set of numbers from which we can choose. The (, ,) term shown in the vertex shows that we have not yet chosen any numbers and that we can choose up to three. The *bound* 5 on the right of the vertex is the target sum required.

At the next level down, the vertices correspond to the subproblems obtained by choosing each of 2, 6 and 3 in term. The sets of numbers from which we can now choose contain only the *other* two numbers in each case. The bounds are reduced to show the *remaining* target sum. For example, the leftmost vertex represents the subproblem of choosing numbers from the set {6, 3} that add up to exactly 3. The bound ∞ at the middle vertex indicates that, having chosen 6, the sum 5 is impossible to attain, and so there is no need to proceed further down this branch — the tree has been pruned.

Note that we avoid duplication of sums of the same sets of numbers by only including vertices that represent adding a number, from the *ordered* set {2, 6, 3}, that lies to the *right* of the last number added. Thus, in particular, once the last number 3 has been chosen, that branch of the tree is terminated.

In this way, for example, we avoid having the sum 2 + 6 + 3 appearing unnecessarily five other times as 2 + 3 + 6, 6 + 2 + 3, 6 + 3 + 2, 3 + 2 + 6 and 3 + 6 + 2.

The vertices at the bottom two levels represent subproblems in a similar way.

For this example, a solution to the original problem is given by any subproblem with bound 0. Once such a bound is reached, that branch is terminated. In this case, there is only one such subproblem, where 2 and 3 have been chosen, adding up to 5 as required. ■

In the above example, the objective is to find numbers that add *exactly* to 5. In other problems that can be solved by the branch-and-bound method, such as finding numbers whose sum is closest to 5, the objective is to find an *optimal* rather than an exact solution. The method can thus be described in general terms as follows.

Of course, any exact solution would be optimal, but the converse does not hold.

Branch-and-bound method

Given a problem that requires the optimization of some objective, determine a **bound** for the problem that corresponds to the objective. If an optimal solution to the problem cannot immediately be found or the problem cannot immediately be shown to have no solution, then break the problem down into a set of two or more subproblems and determine the bound for each.

For each subproblem that is not immediately solved or cannot immediately be shown to have a non-optimal or no solution, break it down further into subproblems and determine their bounds. Repeat this process until each subproblem is solved or has been shown to have a non-optimal or no solution.

An optimal solution to the original problem is given by any subproblem with a 'best' bound.

The solution to each subproblem must also be a solution to the original problem.

The meaning of 'best' will depend on the problem.

A branch-and-bound tree is distinguished in two important ways from a divide-and-conquer tree. First, as we noted above, the branch-and-bound tree is an OR tree, so that the solution to each subproblem is a solution to the original problem. Second, the branch-and-bound tree may be infinite, which cannot be the case for the divide-and-conquer tree.

In the next two subsections, we consider some examples of the use of the branch-and-bound method for solving *optimization* problems. We shall see how the method is used in conjunction with certain *search strategies* that determine the order in which the subproblems are examined.

5.2 Knapsack problem

This problem we consider in this subsection is called the *knapsack problem*, since it may be formulated in the following terms.

Knapsack problem

A hiker is planning a journey, but has a knapsack that can accommodate only a certain total weight. There are a number of items that the hiker wishes to take along, each of which has a particular value for the journey. Which items should be packed so that the total value of the packed items is a maximum, subject to the weight restriction?

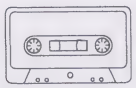
A more practical interpretation of this problem is the following.

Resource selection problem

A company has a certain limited resource that can be used for a number of applications. Each application has a certain value, and uses a certain amount of the resource. Which applications should be chosen so that the greatest total value is obtained from the use of the resource?

The branch-and-bound method can be used for solving such problems, as the audio-tape material associated with this subsection describes.

Now listen to band 1 of Audio-tape 4.



Problem 5.1

A machine in a factory can be used to make any of five items *A*, *B*, *C*, *D* and *E*. The time taken to produce each item, and the value of the item, are shown in the following table:

item	*A*	*B*	*C*	*D*	*E*
production time (in days)	3	7	2	4	4
value	3	14	3	7	8

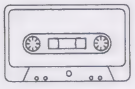
If the machine is available for only 10 days, which of the items should be produced so that the total value is as large as possible?

5.3 Travelling salesman problem Not Assessed

In this subsection, we use the branch-and-bound method to obtain a solution to the travelling salesman problem.

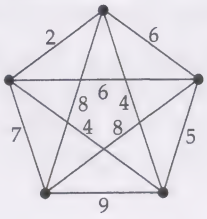
You met the travelling salesman problem in the *Introduction* unit and in *Graphs 2*.

Now listen to band 2 of Audio-tape 4.



Problem 5.2

Use the branch-and-bound method to find a 5-cycle through *A*, *B*, *C*, *D*, *E* with minimum total length for the following weighted graph:



After studying this section, you should be able to:

- explain the terms *state space tree*, *AND tree*, *OR tree* and *AND/OR tree*;
- explain how the *branch-and-bound method* works;
- explain what is meant by a *knapsack*, or *resource allocation, problem*, and use the branch-and-bound method to solve one;
- use the branch-and-bound method to obtain a solution of the travelling salesman problem.

Exercises

Section 1

1.1 Without drawing graphs, show that $O(2^n) \subset O(n!)$.

1.2 Two algorithms for a problem have the following time complexity functions:

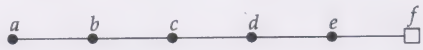
$T_1(n) = 24 \log_3 n + n^2 \log_2 n$

$T_2(n) = 100n^3 + 2^n$

Determine the order of each, and deduce which algorithm is faster.

Section 2

2.1 Consider the following stack s:



Draw the graphs of

(a) PUSH(TOP(s), POP(s));

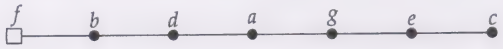
(b) PUSH(f, PUSH(z, POP(s)).

2.2 Consider the following list k:

(a) Determine ITEM(3, k) and ITEM(LENGTH(k), k).

(b) Draw the graph of INSERT(z, 6, k).

2.3 Carry out a bubble sort and a merge sort on the following list:

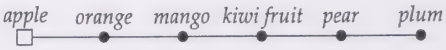


Compare the number of comparisons you made in each case with each other and with the values of the appropriate time complexity functions for lists of 7 items.

$\log_2 7 \approx 2.8$.

Section 3

3.1 Use the GROW-TREE algorithm to construct a binary tree T from the following list:



(a) Is T a balanced tree?

(b) Is T a binary search tree?

(c) Determine:

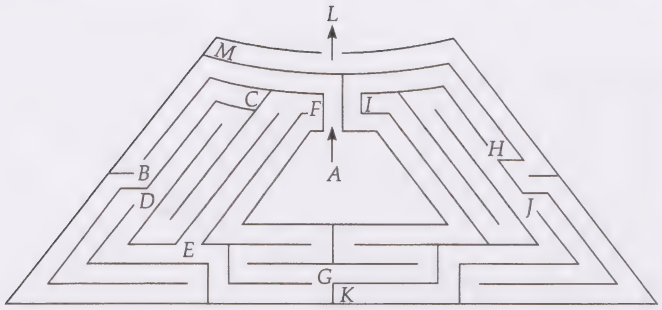
(i) LEFT(T);

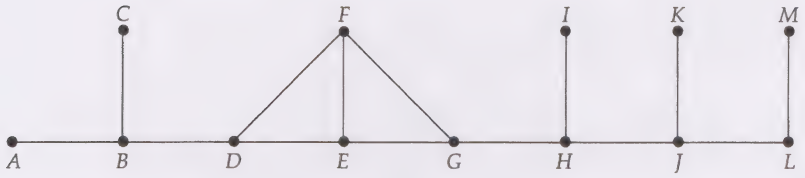
(ii) ROOT(RIGHT(T));

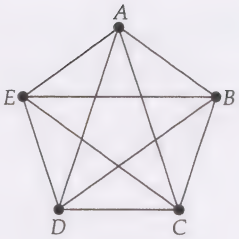
(iii) MAKE-TREE(RIGHT(T), *banana*, LEFT(T)).

3.2 The Hampton Court maze and its graph are shown at the top of the next page.

Use depth-first search to find a route from the centre (A) to the exit (L), based on the rule that, when there is a choice of vertex, choose the one nearest the *end* of the alphabet.



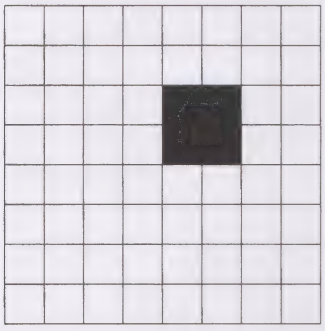




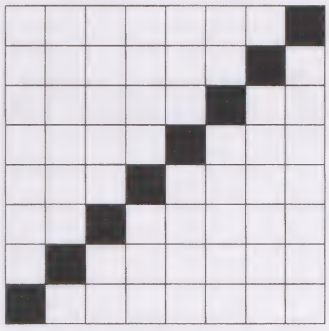
3.3 Find a depth-first search spanning tree and a breadth-first search spanning tree for the graph in the margin, starting from vertex *A*.

Section 4

4.1 Draw the quad trees for the following images on a 4-screen:

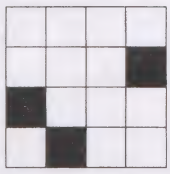


(a)



(b)

4.2 Construct the quad tree for the image in the margin, and hence write down the quad trees for the image after it has been:



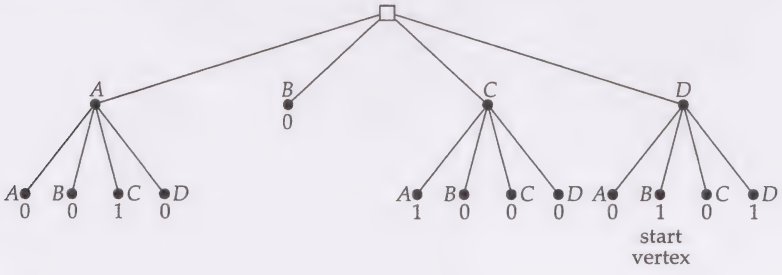
(a) rotated anticlockwise through a right angle;

(b) reflected about a vertical line through the middle of the screen;

(c) first rotated anticlockwise through a right angle and then reflected about a vertical line through the middle of the screen.

Is there a single manipulation that corresponds to the double manipulation in part (c)?

4.3 What changes need to be made to the north neighbour algorithm to turn it into a west neighbour algorithm? Hence find the path taken by the west neighbour algorithm, on the following quad tree, when locating the west neighbour of the pixel represented by the start vertex shown. What colour is the west neighbour of this pixel?



Section 5

5.1 A hiker wishes to take some of the following items on a journey:

item	*A*	*B*	*C*	*D*
weight (lb)	3	5	4	2
value	6	5	3	1

The hiker does not want the contents of his rucksack to weigh more than 9lb. Which items should be taken so that the total value of the contents of the rucksack is as large as possible.

5.2 The distances (in miles) between six Irish cities are shown in the table below:

	Athlone	Dublin	Galway	Limerick	Sligo	Wexford
Athlone	–	78	56	73	71	114
Dublin	78	–	132	121	135	96
Galway	56	132	–	64	85	154
Limerick	73	121	64	–	144	116
Sligo	71	135	85	144	–	185
Wexford	114	96	154	116	185	–

Use the brand-and-bound method to find a shortest route through all six cities for a travelling salesman who wants to visit them all and return to his starting point.

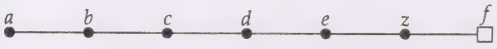
Solutions to the exercises

1.1 For $n = 4$, $2^n = 2^4 = 16 < 24 = 4! = n!$ For each $n > 4$, 2^n is multiplied by 2 to give 2^{n+1} whereas $n!$ is multiplied by $n + 1 > 2$ to give $(n + 1)!$ Therefore $2^{n+1} < (n + 1)!$ for all $n \geq 4$. Hence $O(2^n) \subset O(n!)$.

1.2 The function $T_1(n)$ reduces to $\log_2 n + n^2 \log_2 n$. The function $T_2(n)$ reduces to $n^3 + 2^n$. Since $O(\log_2 n) \subset O(n^2 \log_2 n)$, $T_1(n)$ has order $O(n^2 \log_2 n)$. Since $O(n^3) \subset O(2^n)$, $T_2(n)$ has order $O(2^n)$. Since $O(n^2 \log_2 n) \subset O(2^n)$, the algorithm with time complexity function $T_1(n)$ is faster.

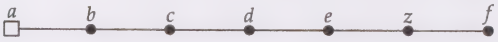
2.1

(a) 

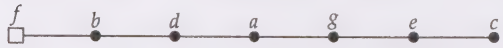
(b) 

2.2

(a) ITEM(3, k) = c and ITEM(LENGTH(k), k) = f.

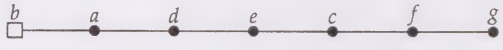
(b) 

2.3 The bubble sort proceeds as follows:

start 

pass 1 

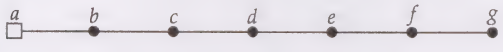
6 comparisons (5 interchanges)

pass 2 

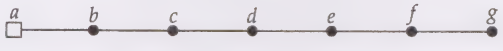
5 comparisons (3 interchanges)

pass 3 

4 comparisons (2 interchanges)

pass 4 

3 comparisons (1 interchange)

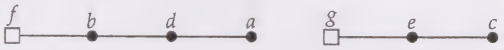
pass 5 

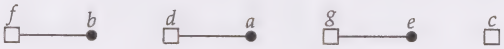
2 comparisons (0 interchanges)

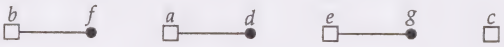
There are no interchanges at pass 5, so there is no need for a sixth pass. The total number of comparisons made is 20.

The merge sort proceeds as follows:

initial list 

first bisection 

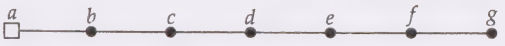
second bisection 

sort 

1 + 1 + 1 + 0 = 3 comparisons

first merge 

3 + 1 = 4 comparisons

second merge 

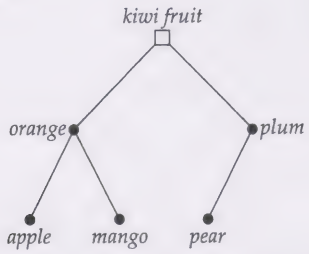
6 comparisons

The total number of comparisons made is 13.

For a bubble sort $T(n) = n(n-1)/2$, which in this case gives $T(7) = 21$. For a merge sort $T(n) = n\log_2 n$, which in this case gives $T(7) \approx 7 \times 2.8 = 19.6 \approx 20$.

So, for a list of 7 items, in the *worst* case the number of comparisons needed by the two algorithms is roughly the same. But in *this* case, whereas the bubble sort needed nearly all of the possible comparisons, the merge sort needed only about two-thirds of them.

3.1 The binary tree T is:



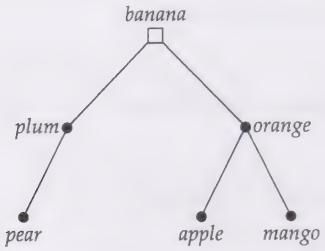
(a) T is balanced — GROW-TREE always produces balanced trees.

(b) T is not a binary search tree, as the list was not ordered to begin with. So, for example, *orange*, which is larger than *kiwi fruit*, appears below and to the left of *kiwi fruit*, contrary to the definition of a binary search tree.

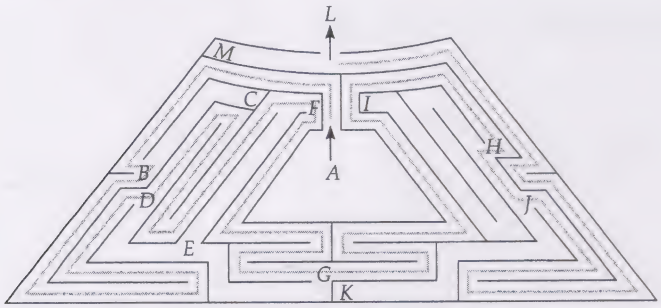
(c) (i) LEFT(T) is:

(ii) ROOT(RIGHT(T)) = *plum*

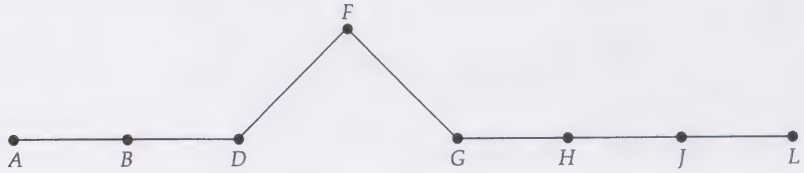
(iii) MAKE-TREE(RIGHT(T), *banana*, LEFT(T)) is:



3.2 Depth-first search gives the following route out of the maze:

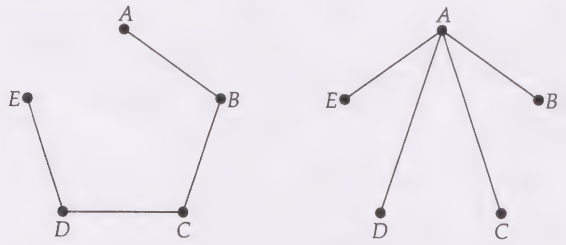


The corresponding graph is:

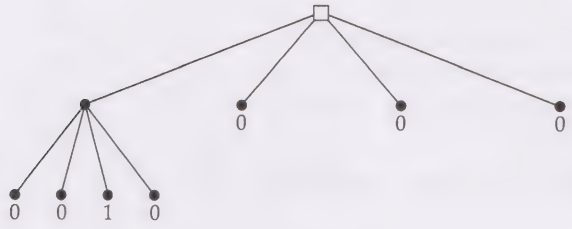


(Notice that this is not a spanning tree, since in this case the search stops when we reach L.)

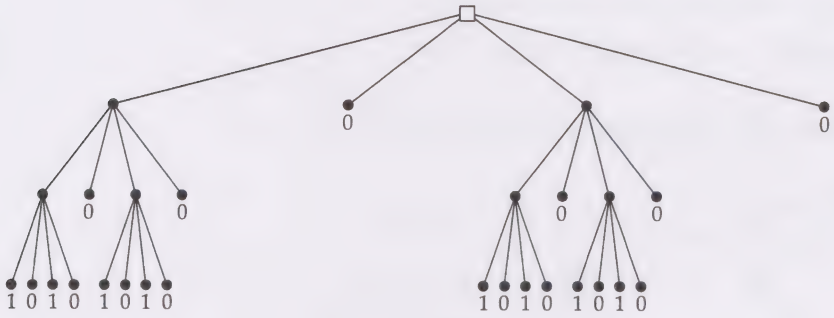
3.3 There are several possible depth-first search spanning trees, one of which is shown on the left below. There is just *one* breadth-first search spanning tree, shown on the right below.



4.1

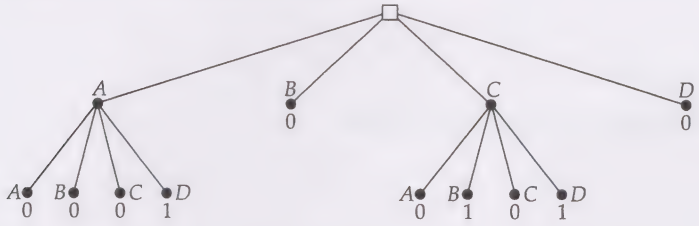


(a)

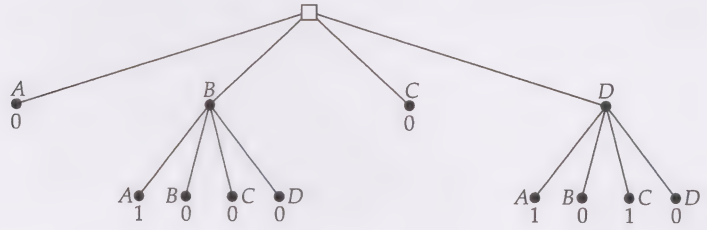


(b)

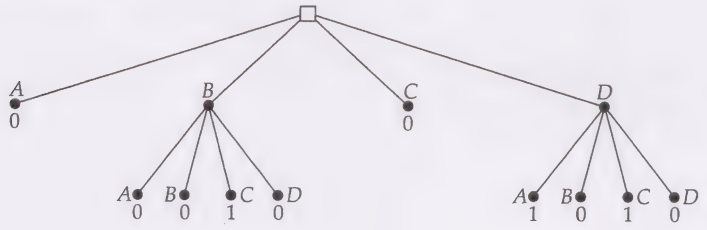
4.2 The quad tree for original image is:



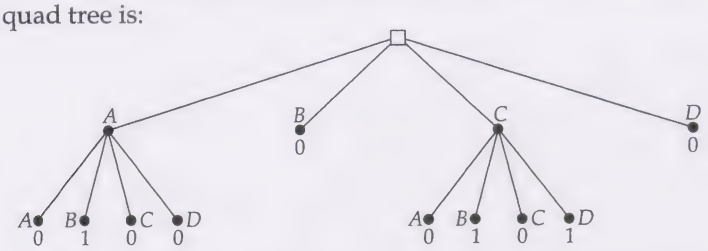
(a) Rotation anticlockwise through a right angle uses the scheme $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. Hence the new quad tree is:



(b) Reflection about a vertical line uses the scheme $A \leftrightarrow B$ and $C \leftrightarrow D$. Hence the new quad tree is:

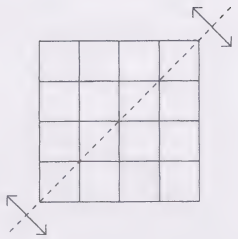


(c) The combination of the two manipulations uses the scheme $A \to B \to C \to D \to A$ followed by the scheme $A \leftrightarrow B$ and $C \leftrightarrow D$. Hence the new quad tree is:



The scheme $A \to B \to C \to D \to A$ followed by $A \leftrightarrow B$ and $C \leftrightarrow D$ gives:

$A \to B \to A$ $\quad B \to C \to D$ $\quad C \to D \to C$ $\quad D \to A \to B$



So the overall effect is just $B \leftrightarrow D$, and hence the double manipulation in part (c) corresponds to the single manipulation of reflection in the diagonal through A and C — that is, reflection in the top-right-to-bottom-left diagonal.

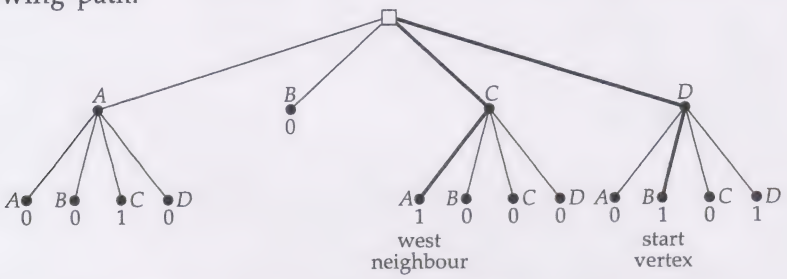
4.3 To change the north neighbour algorithm to a west neighbour algorithm, we must:

STEP 2 Change 'lower (C or D) vertex' to 'east (A or D) vertex' twice.

STEP 3 Remember that in this case A and B are opposites, as are C and D.

Also, of course, we must replace the words 'north neighbour' by 'west neighbour' in Steps 2 and 3.

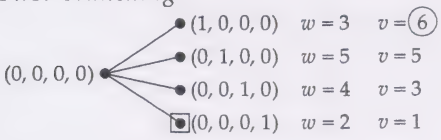
Applying the west neighbour algorithm to the given quad tree gives the following path:



Hence the west neighbour of the given pixel is black.

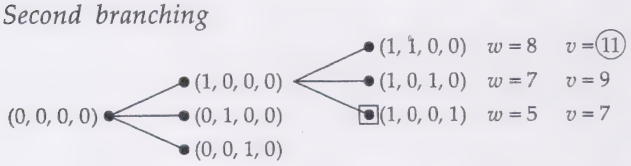
5.1 Using the branch-and-bound method from the tape, we carry out the branching as follows:

First branching



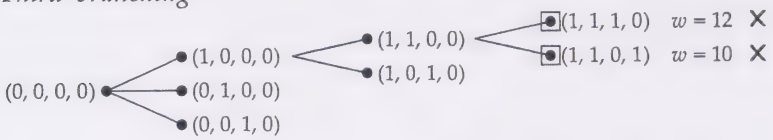
Store: (1, 0, 0, 0), $v = 6$.

Second branching



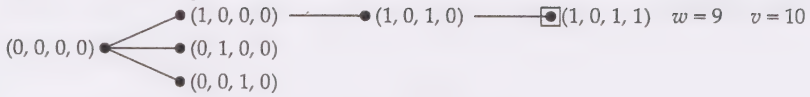
Store: (1, 1, 0, 0), $v = 11$.

Third branching



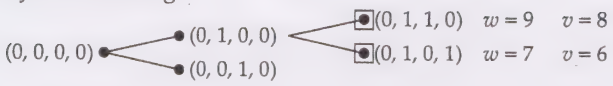
Store: unchanged.

Fourth branching



Store: unchanged.

Fifth branching



Store: unchanged.

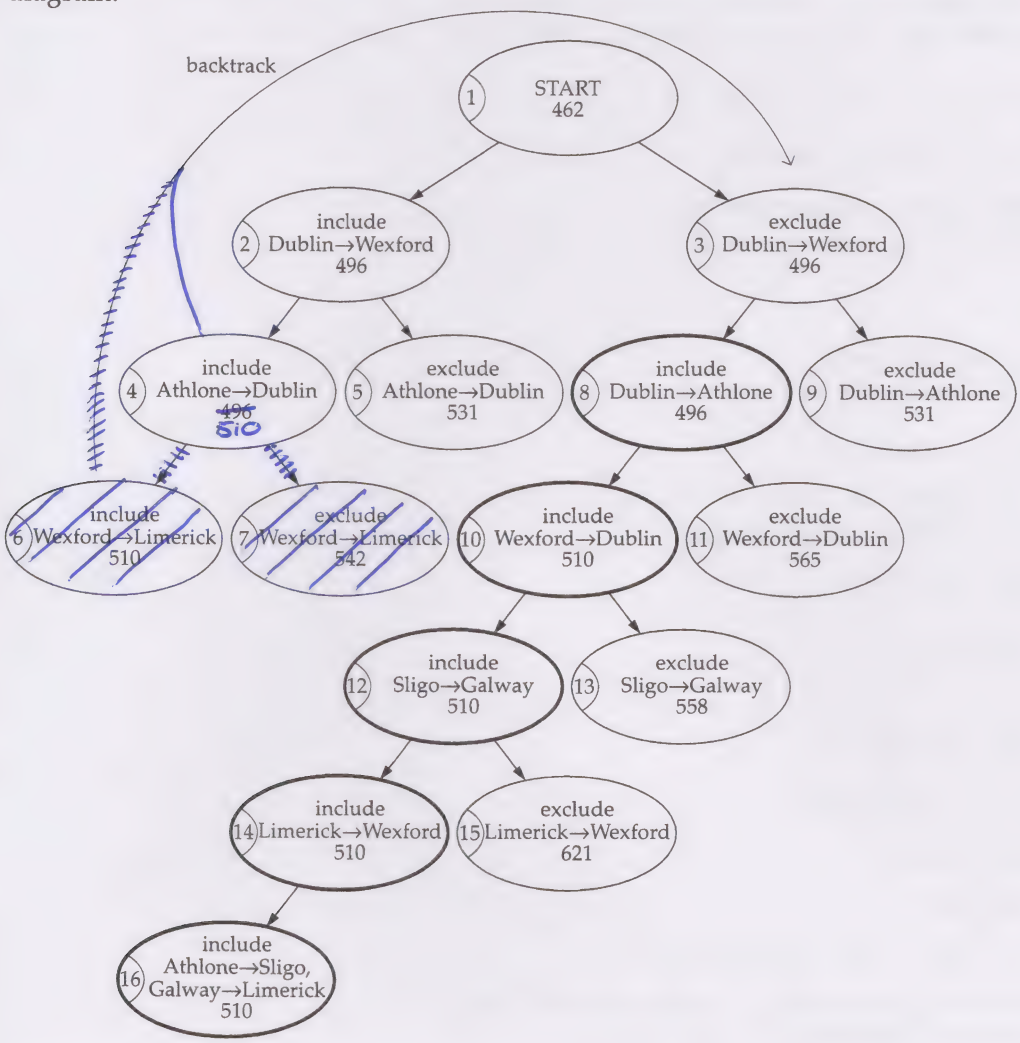
Sixth branching

Store: unchanged.

No further branches to explore.

The solution vector is (1, 1, 0, 0), so that the hiker should take items *A* and *B*, of total weight 8 lb and total value 11.

5.2 One possible solution process is summarized in the following diagram:



Thus one possible route for the travelling salesman is

Dublin → Athlone → Sligo → Galway → Limerick → Wexford → Dublin

covering a total of 510 miles.

Solutions to the problems

Solution 1.1

n	5	10	20
$T_1(n)$	100	100	100
$T_2(n)$	50	100	200
$T_3(n)$	25	100	400

When $n < 10$, the algorithm with time complexity function T_1 is slowest and that with time complexity function T_3 is fastest;

when $n = 10$, all three algorithms take the same time;

when $n > 10$, the algorithm with time complexity function T_1 is fastest and that with time complexity function T_3 is slowest.

Solution 1.2

(a) $2n^2 + 4n + 3 > 4n > c.1$ for some constant c

whenever $4n > c$, i.e. whenever $n > c/4$. Hence, however large c is, inequality 1.1 does not hold for $n > c/4$, and so $T(n)$ is not dominated by 1.

Similarly, for $n > 0$,

$$2n^2 + 4n + 3 > 2n^2 > c.n \quad \text{for some constant } c$$

whenever $n > c/2$. Hence, $T(n)$ is not dominated by n.

However,

$$\begin{aligned} T(n) &= 2n^2 + 4n + 3 \\ &\leq 2n^2 + 4n^2 + 3n^2 \quad \text{for } n \geq 1 \\ &= (2+4+3)n^2 \\ &\leq (2+4+3)n^3 \quad \text{for } n \geq 1. \end{aligned}$$

Hence inequality 1.1 holds for $T(n)$ with $g(n) = n^2$ or n^3, $c = 2 + 4 + 3 = 9$ and $N = 1$. So $T(n)$ is dominated by n^2 and n^3.

(b) $1 \leq \frac{1}{3}.(2n^2 + 4n + 3)$ for all $n \geq 0$

Hence $2n^2 + 4n + 3$ dominates 1.

Similarly:

$$n \leq \tfrac{1}{4}.(2n^2 + 4n + 3) \quad \text{for all } n$$

$$n^2 \leq \tfrac{1}{2}.(2n^2 + 4n + 3) \quad \text{for all } n \geq 0$$

So $2n^2 + 4n + 3$ dominates n and n^2.

However, for $n > 9c$ we have

$$n^3 > c.9n^2 = c.(2+4+3)n^2 \geq c.(2n^2 + 4n + 3) \quad \text{for } n \geq 1.$$

So, however large we make c, inequality 1.1 does not hold when $n > 9c$. So $2n^2 + 4n + 3$ does not dominate n^3.

Solution 1.3

For $n > 2$, we know that $1 < \log_2 n < n$. Multiplying through by n (> 2), we obtain $n < n\log_2 n < n^2$ for all $n > 2$. We can therefore deduce that the set $O(n\log_2 n)$ is located in the hierarchy as follows:

$$\ldots \subset O(n) \subset O(n\log_2 n) \subset O(n^2) \subset \ldots$$

Solution 1.4

Using the procedure, $T_1(n) = 9n^3 + 5n + 3\log_{10}n$ reduces to $n^3 + n + \log_2 n$.
Since $O(\log_2 n) \subset O(n) \subset O(n^3)$, $T_1(n)$ has order $O(n^3)$.

Similarly, $T_2(n) = 1000\log_2 n + 2n^2 + 100$ reduces to $\log_2 n + n^2 + 1$.
Since $O(1) \subset O(\log_2 n) \subset O(n^2)$, $T_2(n)$ has order $O(n^2)$.

Since $O(n^2) \subset O(n^3)$, the algorithm corresponding to $T_2(n)$ is faster, for large n.

Solution 1.5

(a) For $n > 1$, we know from Solution 1.3 that $n < n\log_2 n < n^2$. Mutiplying through by n (> 1), we obtain $n^2 < n^2\log_2 n < n^3$ for all $n > 1$. We deduce that the set $O(n^2\log_2 n)$ is located in the order hierarchy as follows:

$$\ldots \subset O(n^2) \subset O(n^2\log_2 n) \subset O(n^3) \subset \ldots$$

(b) Using the procedure, the two time complexity functions reduce to $n^2 + \log_2 n$ and $n^2\log_2 n$. Since $O(\log_2 n) \subset O(n^2)$, the first has order $O(n^2)$; the second has order $O(n^2\log_2 n)$. Since $O(n^2) \subset O(n^2\log_2 n)$, the algorithm with time complexity function $2n^2 + \log_3 n$ is faster.

Solution 2.1

The given stack is:

4	
3	*aardvark*
2	*tiger*
1	*lion*
TOP	3

(a) The tape for the stack PUSH(*iguana*, *s*) is:

4	*iguana*
3	*aardvark*
2	*tiger*
1	*lion*
TOP	4

(b) The tape for the stack POP(*s*) is:

4	
3	
2	*tiger*
1	*lion*
TOP	2

(c) The tape for the stack POP(POP(*s*)) is:

4	
3	
2	
1	*lion*
TOP	1

Solution 2.2

(a) The tape for the stack PUSH(*iguana*, POP(*s*)) is:

4	
3	*iguana*
2	*tiger*
1	*lion*
TOP	3

(b) TOP(POP(*s*)) = *tiger*

(c) DEPTH(POP(*s*)) = 2

(d) TOP(POP(POP(*s*)) gives the third item from the top, namely *lion*.

Solution 2.3

TOP(*s*) corresponds to ITEM(LENGTH(*k*), *k*).

POP(*s*) corresponds to DELETE(LENGTH(*k*), *k*).

PUSH(item, *s*) corresponds to INSERT(item, LENGTH(*k*) + 1, *k*).

Solution 2.4

Replace the address 3 in cell 2 by the START address of list *k'* (namely 6), and store that forwarding address 3. Replace the value 0 in the end cell of list *k'* (cell 9) by the stored forwarding address 3. Replace the value *k* in the NAME cell for list *k* by *k''* and the value 2 in the LENGTH cell by 4 (= LENGTH(*k*) + LENGTH(*k'*)).

The forwarding address would be stored as the only item in temporary stack, say.

10	
9	0
8	*zebra*
7	8
6	*aardvark*
5	
4	0
3	*tiger*
2	3
1	*lion*
NAME	*k*
START	1
LENGTH	2
NAME	*k'*
START	6
LENGTH	2

separate lists *k* and *k'*

10	
9	3
8	*zebra*
7	8
6	*aardvark*
5	
4	0
3	*tiger*
2	6
1	*lion*
NAME	*k''*
START	1
LENGTH	4

k' inserted into *k* to give new list *k''*

Solution 2.5

(a) $T(3) = 2 + T(2)$

$= 2 + (2 + T(1))$

$= 2 + (2 + (2 + T(0)))$

$= 2 + (2 + (2 + 1))$

$= 7.$

(b) $T(n) = 2 + T(n-1)$

$= 2 + 2 + T(n-2)$

$= 2 + 2 + 2 + T(n-3)$

$\vdots$

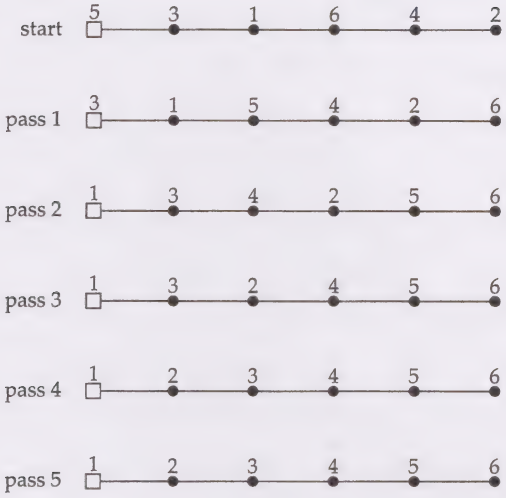
$= 2 + 2 + 2 + \ldots + 2 + T(1)$

$= 2 + 2 + 2 + \ldots + 2 + 2 + T(0)$

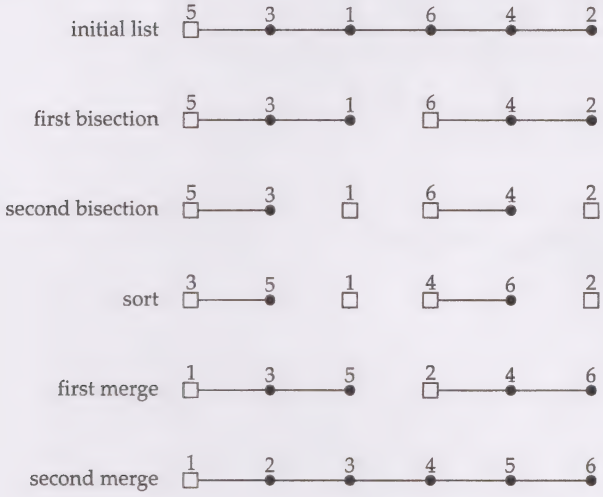
$= \underbrace{2 + 2 + 2 + \ldots + 2 + 2}_{n \text{ terms}} + 1$

$= 2n + 1.$

Solution 2.6



Solution 2.7



Solution 2.8

In the worst case, for a list of 32 items, the bubble sort algorithm takes

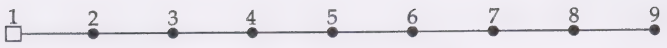
$32(32-1)/2 = 496$ time units

whereas the merge sort algorithm takes

$32 \log_2 32 = 32 \times 5 = 160$ time units.

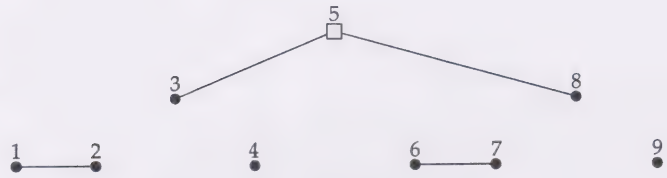
Solution 3.1

initial list

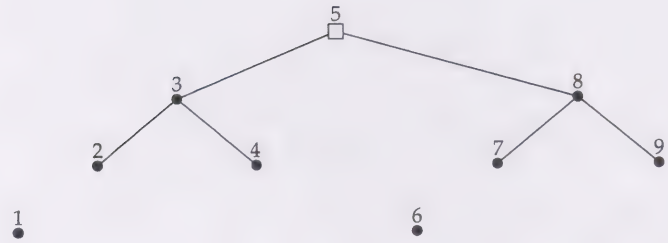


first iteration

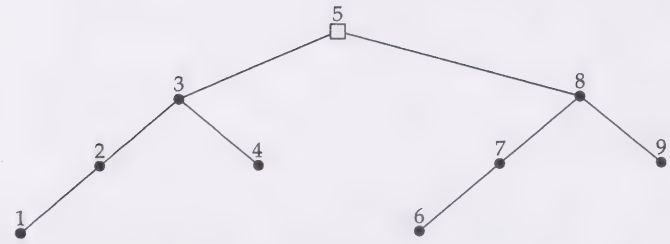
second iteration



third iteration



fouth iteration



Solution 3.2

(a) No, it is not balanced. The height of the left subtree of the root vertex is 1, while that of the right subtree is 3.

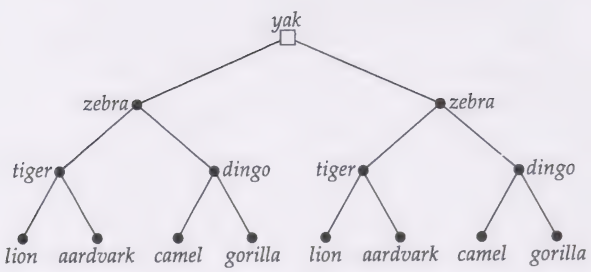
(b) Yes, it is balanced.

Solution 3.3

(a)

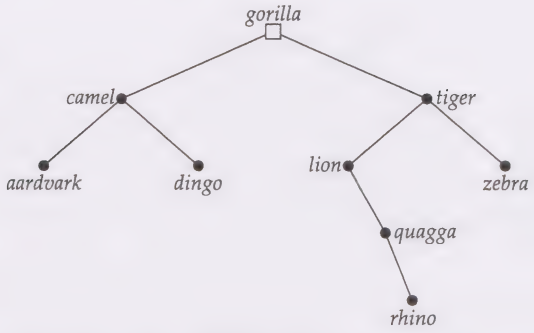
(b) *dingo*

(c)



Solution 3.4

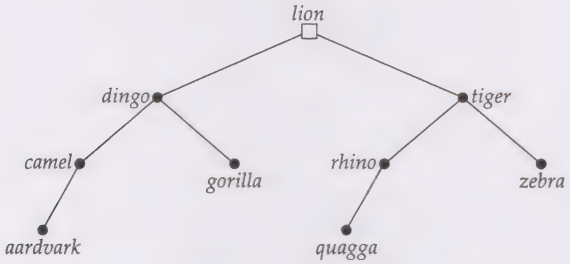
(a) The resulting binary search tree is



This is unbalanced since the heights of the left and right subtrees of *gorilla*, *tiger* and *lion* all differ by 2.

(b) The new ordered list is

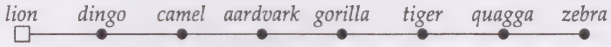
The binary search tree obtained by applying GROW-TREE to this list is:



This is balanced of height 4, whereas the binary tree in part (a) is unbalanced of height 5.

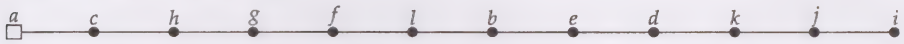
Solution 3.5

The resulting list is:



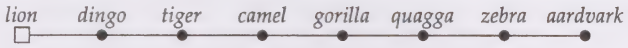
Solution 3.6

The resulting list is:



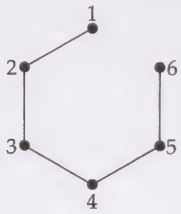
Solution 3.7

The resulting list is:

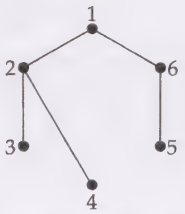


Solution 3.8

Two possible such trees are:



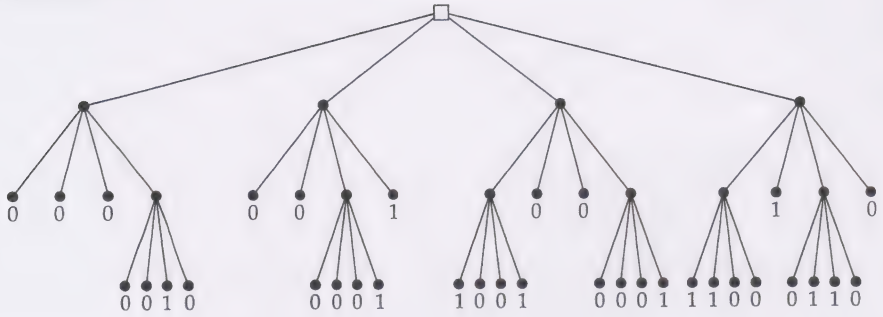
depth-first search spanning tree



breadth-first search spanning tree

There are *several* possible depth-first search spanning trees and *two* possible breadth-first search spanning trees, starting from vertex 1.

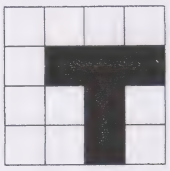
Solution 4.1



Solution 4.2

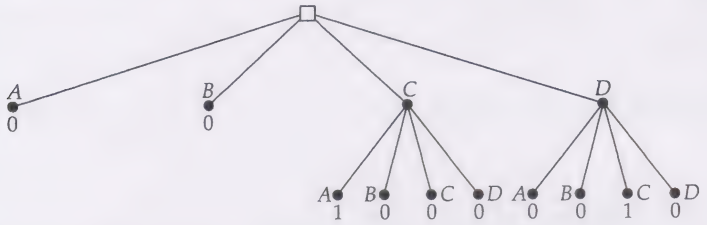
A full (unpruned) quad tree is needed when all the end-vertices of the quad tree must occur at the lowest level possible (i.e. at the level of the single pixel). So, in each of the 2×2 quadrants, there must be at least one pixel of a different colour from the rest.

Solution 4.3

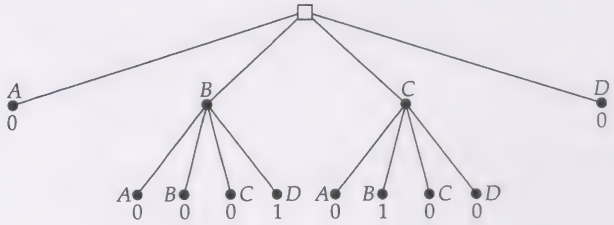


Solution 4.4

For the given image, the quad tree is:



Clockwise rotation through a right angle requires movement to the left, using the cycle $A \rightarrow D \rightarrow C \rightarrow B \rightarrow A$, to give the quad tree:



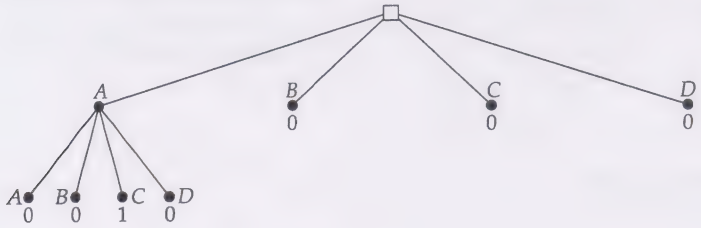
Solution 4.5

To produce a rotation through two right angles we need to displace what is stored at each vertex two places, either to the left or to the right, using the cycles

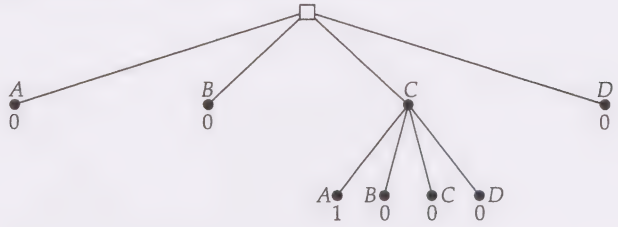
$$A \rightarrow C \rightarrow A \quad \text{and} \quad B \rightarrow D \rightarrow B.$$

Solution 4.6

The quad tree for the original image is:

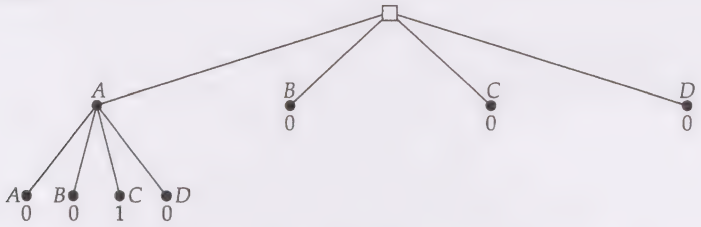


After the interchange $A \leftrightarrow C$, we have the tree:

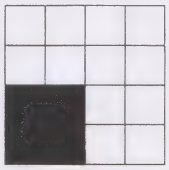


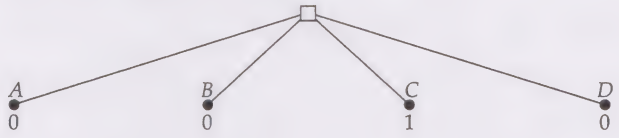
Solution 4.7

The quad tree for the original image is the same as that for the image in Problem 4.6:



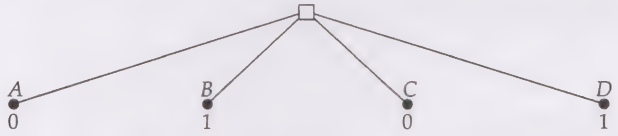
Detaching the subtree from the A-vertex at level 1 gives the following quad tree, and its corresponding image:



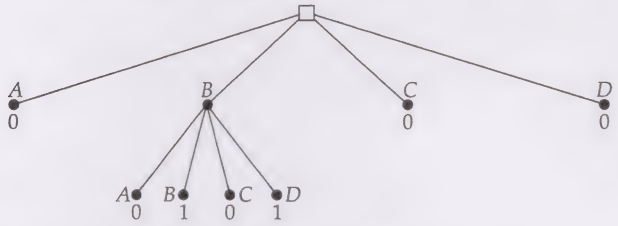


Solution 4.8

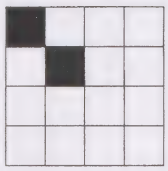
The quad tree for the original image is:



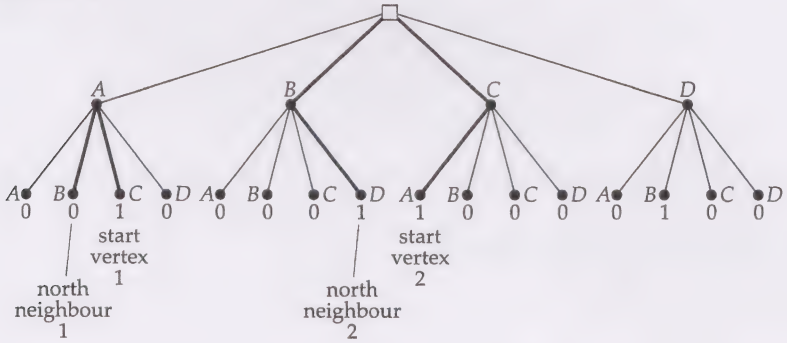
We attach this as a subtree to the B-vertex at level 1 in a new tree:



The corresponding screen image is shown in the margin.



Solution 4.9



The paths traced by the north neighbour algorithm in each case are shown above. Thus the north neighbour of pixel 1 is white, and the north neighbour of pixel 2 is black.

Solution 5.1

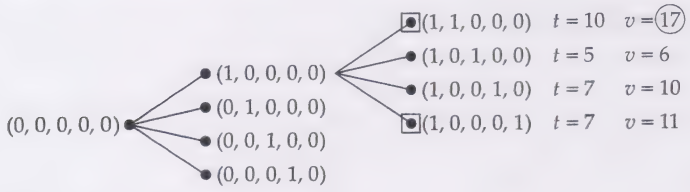
item	A	B	C	D	E
production time (in days) t	3	7	2	4	4
value v	3	14	3	7	8

Maximum total production time = 10 days.

First branching: from zero solution vector (0, 0, 0, 0, 0) with $v = 0$.

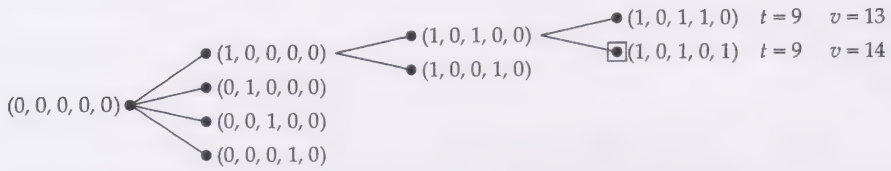
Store: (0, 1, 0, 0, 0), $v = 14$.

Second branching: from (1, 0, 0, 0, 0).



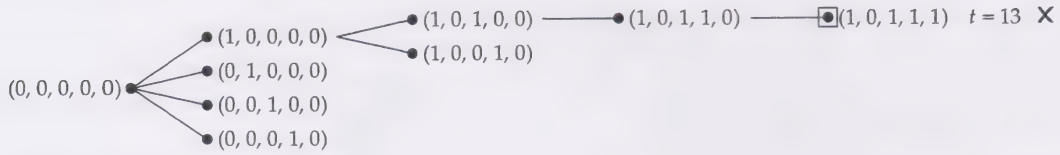
Store: (1, 1, 0, 0, 0), $v = 17$.

Third branching: from (1, 0, 1, 0, 0).



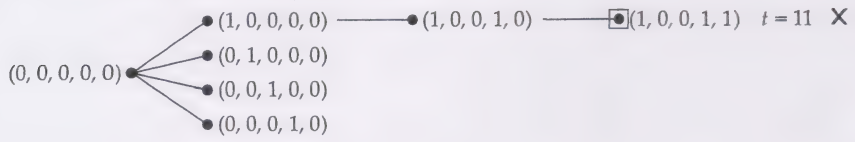
Store: unchanged.

Fourth branching: from (1, 0, 1, 1, 0).



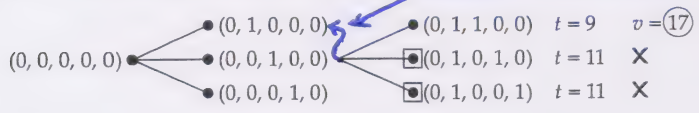
Store: unchanged.

Fifth branching: from (1, 0, 0, 1, 0).



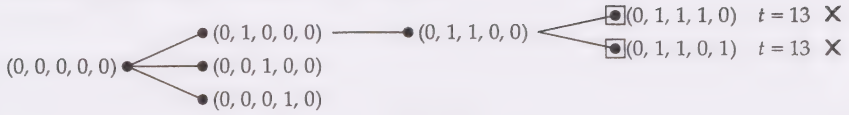
Store: unchanged.

Sixth branching: from (0, 1, 0, 0, 0). should emanate from here (Empty)



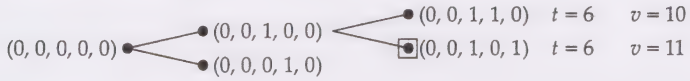
Store: (1, 1, 0, 0, 0), (0, 1, 1, 0, 0), $v = 17$.

Seventh branching: from (0, 1, 1, 0, 0).



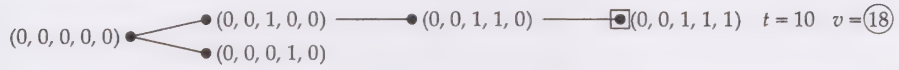
Store: unchanged.

Eighth branching: from (0, 0, 1, 0, 0).



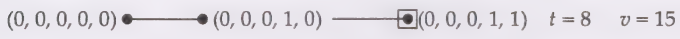
Store: unchanged.

Ninth branching: from (0, 0, 1, 1, 0).



Store: (0, 0, 1, 1, 1), $v = 18$.

Tenth branching: from (0, 0, 0, 1, 0).



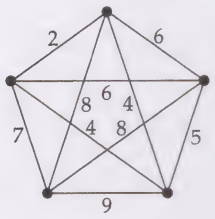
Store: unchanged.

No further branches to explore.

The solution vector is (0, 0, 1, 1, 1), so that items *C*, *D* and *E* should be produced, with value 18.

Solution 5.2

The weighted graph and corresponding table of weights are:



	A	*B*	*C*	*D*	*E*
A	–	6	4	8	2
B	6	–	5	8	6
C	4	5	–	9	4
D	8	8	9	–	7
E	2	6	4	7	–

Initial lower bound

		A	*B*	*C*	*D*	*E*
2	*A*	–	4	2	6	0
5	*B*	1	–	0	3	1
4	*C*	0	1	–	5	0
7	*D*	1	1	2	–	0
2	*E*	0	4	2	5	–

→

		0	1	0	3	0
		A	*B*	*C*	*D*	*E*
2	*A*	–	3	2	3	0
5	*B*	1	–	0	0	1
4	*C*	0	0	–	2	0
7	*D*	1	0	2	–	0
2	*E*	0	3	2	2	–

lower bound = (2 + 5 + 4 + 7 + 2) + (1 + 3) = 24

First branching

1	*A*	*B*	*C*	*D*	*E*
A	–	3	2	3	0^2
B	1	–	0^2	0^2	1
C	0^0	0^0	–	2	0^0
D	1	0^0	2	–	0^0
E	$0^{\cancel{0}2}$	3	2	2	–

include *AE*
reduce row *E* by 2

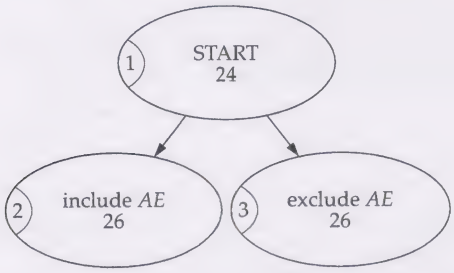
	2	*A*	*B*	*C*	*D*
0	*B*	1	–	0	0
0	*C*	0	0	–	2
0	*D*	1	0	2	–
2	*E*	X	1	0	0

new lower bound = 24 + 2 = 26

exclude *AE*
reduce row *A* by 2

	3	*A*	*B*	*C*	*D*	*E*
2	*A*	–	1	0	1	X
0	*B*	1	–	0	0	1
0	*C*	0	0	–	2	0
0	*D*	1	0	2	–	0
0	*E*	0	3	2	2	–

new lower bound = 24 + 2 + 26



Second branching

Since both new lower bounds are the same, we can choose either branch.
Let us choose to include *AE*.

2	*A*	*B*	*C*	*D*
B	1	–	0^0	0^0
C	0^1	0^0	–	2
D	1	0^1	2	–
E	X	1	0^0	0^0

include *CA*
(so exclude *EC*)

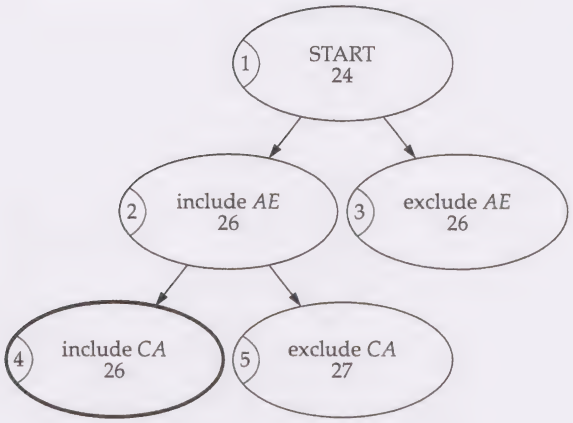
4	*B*	*C*	*D*
B	–	0	0
D	0	2	–
E	1	X	0

lower bound remains = 26

exclude *CA*
reduce column *A* by 1

	1	0	0	0
5	*A*	*B*	*C*	*D*
B	0	–	0	0
C	X	0	–	2
D	0	0	2	–
E	X	1	0	0

new lower bound = 26 + 1 = 27



Third branching

We choose to continue down current branch, by including CA.

4	B	C	D
B	–	0^2	0
D	0^3	2	–
E	1	X	0^1

include DB
(so exclude BD)

6	C	D
B	0	X
E	X	0

lower bound remains = 26

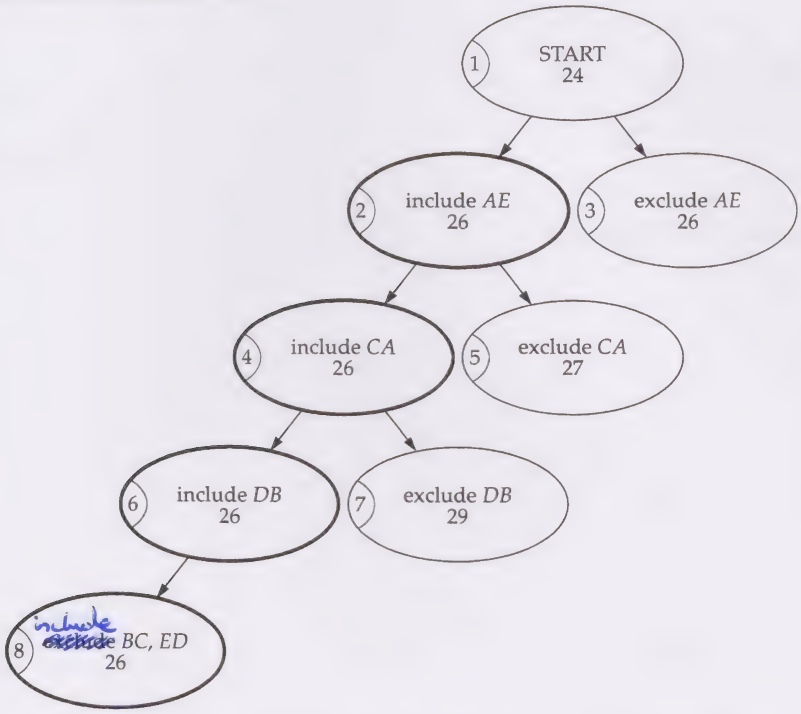
exclude DB
reduce row D by 2
reduced column B by 1

		1	0	0
	7	B	C	$\not{B}D$
0	B	–	0	0
2	D	X	0	–
0	E	0	X	0

new lower bound = 26 + 3 = 29

include BC, ED

lower bound remains = 26



We have a 5-cycle with edges AE, ED, DB, BC, CA and of length 26. No other branch of the tree can lead to a shorter 5-cycle. So we have a solution to the problem.

If you chose to exclude AE on the second branching, your solution should have progressed as follows:

Second branching

3	A	B	C	D	E
A	–	1	0^1	1	X
B	1	–	0^0	0^1	1
C	0^0	0^0	–	2	0^0
D	1	0^0	2	–	0^0
E	0^0	3	2	2	–

include AC
(so exclude CA)

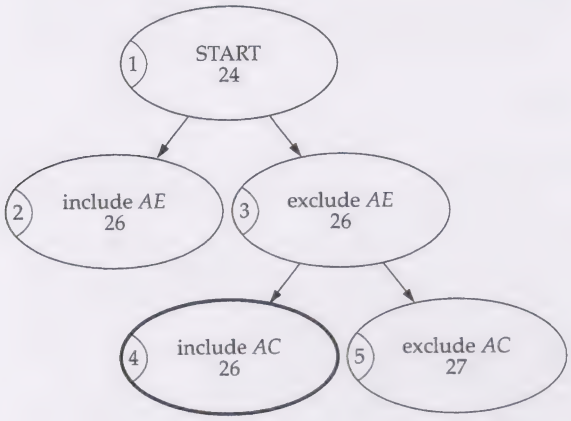
4	A	B	D	E
B	1	–	0	1
C	X	0	2	0
D	1	0	–	0
E	0	3	2	0

lower bound remains = 26

exclude AC
reduce row A by 1

	5	A	B	C	D	E
1	A	–	0	X	0	X
0	B	1	–	0	0	1
0	C	0	0	–	2	0
0	D	1	0	2	–	0
0	E	0	3	2	2	–

new lower bound = 26 + 1 = 27



Third branching

We choose to continue down the current branch, by including AC.

4	A	B	D	E
B	1	–	0^3	1
C	X	0^0	2	0^0
D	1	0^0	–	0^0
E	0^1	3	2	0^0

include BD
(so exclude DB)

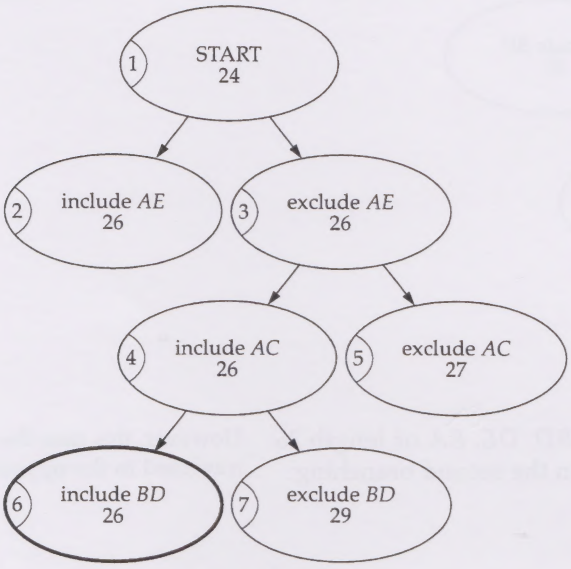
6	A	B	E
C	X	0	0
D	1	X	0
E	0	3	0

lower bound remains = 26

exclude BD
reduce row B by 1
reduced column D by 2

		0	0	2	0
	7	A	B	D	E
1	B	0	–	X	0
0	C	X	0	0	0
0	D	1	0	–	0
0	E	0	3	0	0

new lower bound = 26 + 3 = 29



Fourth branching

We choose to continue down the current branch, by including BD.

6	A	B	E
C	X	0^3	0^2
D	1	X	0^1
E	0^1	3	0^0

include CB
(so exclude BC, DA)

8	A	E
D	X	0
E	0	0

lower bound remains = 26

exclude CB
reduce column B by 3

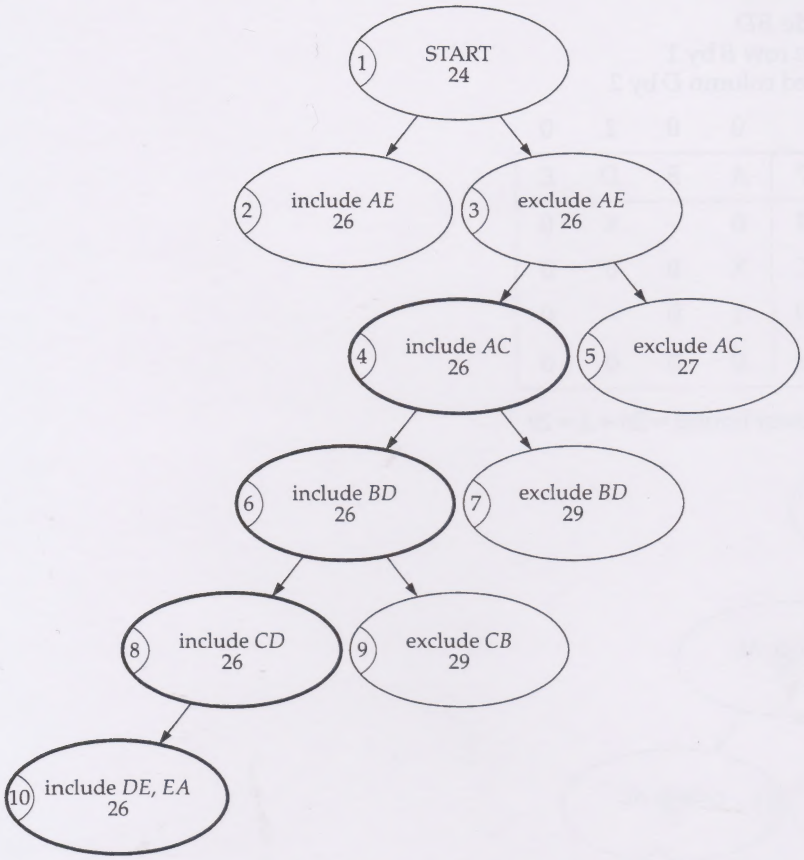
	0	3	0
9	A	B	E
C	X	X	0
D	1	X	0
E	0	0	0

new lower bound = 26 + 3 = 29

include DE
(so exclude ED)

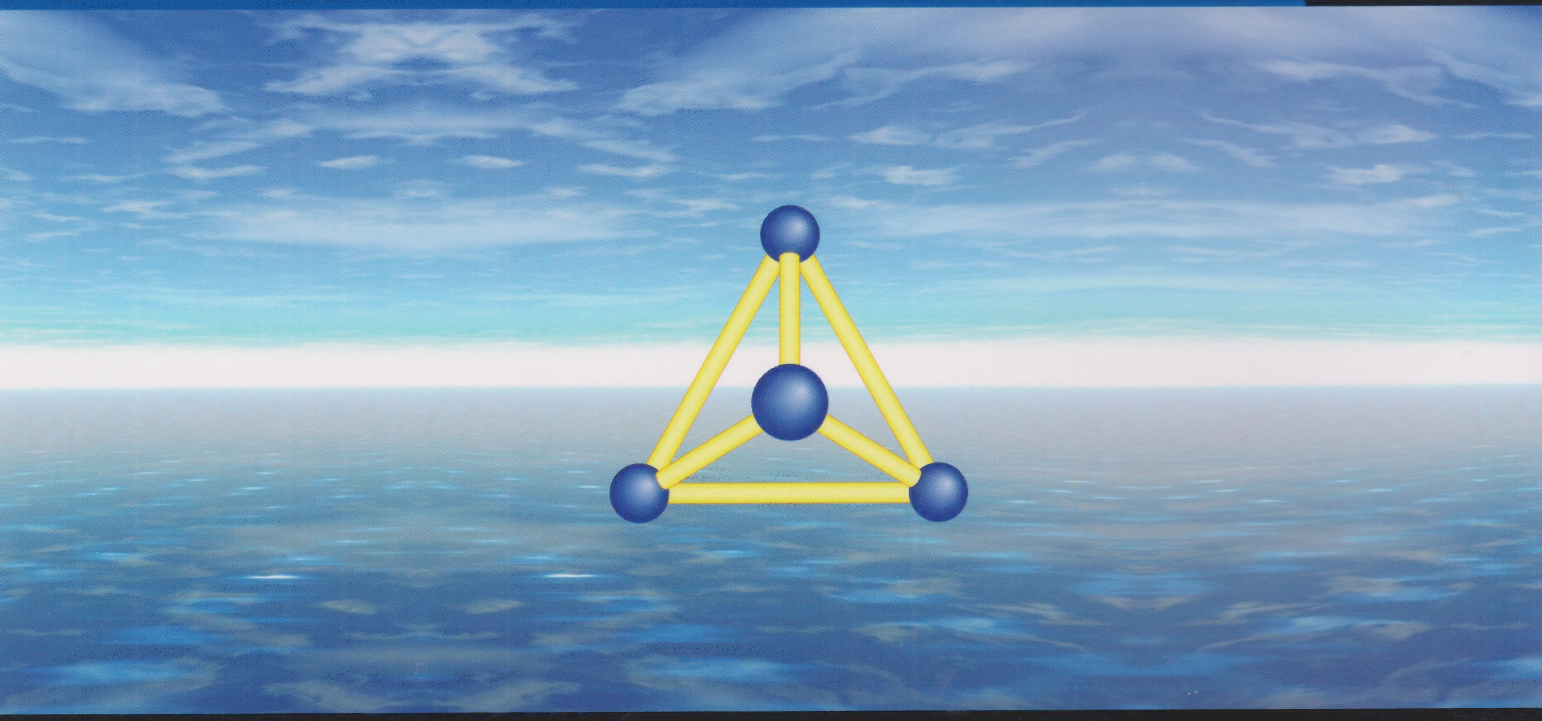
include EA

lower bound remains = 26



So we have the same 5-cycle with edges AC, CB, BD, DE, EA of length 26 as we would have if we had chosen to include AE on the second branching. However, this time the cycle is traversed in the opposite direction.

Index



MT365 Graphs, networks and design

The Open University

MT365 Graphs 4
ISBN 978 0 7492 5425 4

9 780749 254254